

MULTI: Building Applications for Embedded PowerPC



Green Hills Software, Inc.

**30 West Sola Street
Santa Barbara, California 93101
USA**

**Tel: 805-965-6044
Fax: 805-965-6343
www.ghs.com**

DISCLAIMER

GREEN HILLS SOFTWARE, INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software, Inc. to notify any person of such revision or changes.

Copyright © 1983-2010 by Green Hills Software, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from Green Hills Software, Inc.

Green Hills, the Green Hills logo, CodeBalance, GMART, GSTART, INTEGRITY, MULTI, and Slingshot are registered trademarks of Green Hills Software, Inc. AdaMULTI, Built with INTEGRITY, EventAnalyzer, G-Cover, GHnet, GHnetLite, Green Hills Probe, Integrate, ISIM, u-velOSity, PathAnalyzer, Quick Start, ResourceAnalyzer, Safety Critical Products, SuperTrace Probe, TimeMachine, TotalDeveloper, DoubleCheck, and velOSity are trademarks of Green Hills Software, Inc.

All other company, product, or service names mentioned in this book may be trademarks or service marks of their respective owners.

PubID: 10684
September 1, 2015

Contents

	Preface	xix
	About This Book	xx
	The MULTI 5.3 Document Set	xxi
	Conventions Used in the MULTI Document Set	xxii
1	Introduction	1
	The MULTI Integrated Development Environment	2
	MULTI Launcher	2
	Editing Tools	2
	Building Tools	2
	Debugging Tools	3
	Administrative Tools	3
	The MULTI Launcher	4
	MULTI Workspaces	5
	The MULTI Toolchain	5
	The MULTI Project Manager	5
	The C and C++ Compiler Driver	7
	The Optimizing Compilers	7
	The asppc Assembler	8
	The ax Librarian	8
	The elxr Linker	8
	Optimized Libraries and Header Files	9
	Utility Programs	9
	Accessing Online Help	10
	Full Online Manuals	10
	Context-Sensitive Help	10
	Command, Configuration Option, and Keyword Help	10
	The Help Viewer Window	11
	Navigating Manuals in the Help Viewer	13

Contents

Part I	Using the MULTI Builder and Compiler Drivers	
2	The MULTI Project Manager	19
	Starting the MULTI Project Manager	22
	Creating A New Project	22
	Using the Project Wizard	23
	Adding Executables and Examples to the Top Project	26
	Opening the Program in the MULTI Debugger	26
	Managing Your Project	27
	Top Projects	28
	Target Resources Project	28
	Adding New Items to Your Project	29
	Adding Existing Files To Your Project	33
	Configuring Existing Items	33
	Moving Items in the Project Hierarchy	35
	Common Scenarios	36
	Green Hills Project Files	38
	File Types	39
	Automatically Including Files	41
	Building Your Project	42
	Using the File Shortcut Bar	43
	Navigating the Project Tab	45
	Navigating the Memory Layout Tab	45
	Searching for Project Files	46
	Choosing a Build Mode	47
	Building Platform-Specific Programs from the Same Source Files	48
	Setting Builder Options	49
	The Build Options Window	50
	Setting Build Macros	60
	Importing Environment Variables	61
	Inheriting Options from Parent Build Files	62
	Setting Options for Projects and Subprojects	62
	Setting Options for Source Files	63
	Working with Linker Directives Files	63
3	The Project Manager GUI Reference	65
	The File Menu	66

Contents

	The Edit Menu	67
	The Target Selector Dialog Box	71
	The View Menu	72
	The Build Menu	73
	The Advanced Build Dialog Box	74
	The Connect Menu	76
	The Debug Menu	77
	The Tools Menu	78
	The Utility Program Launcher Dialog Box	79
	The Windows Menu	80
	The Help Menu	80
4	The Compiler Driver	83
	The Compiler Driver Syntax	85
	Building an Executable from C or C++ Source Files	86
	Working with Input Files	87
	Recognized Input File Types	87
	Passing Multiple Input File Types to the Driver	88
	Passing Linker Directives Files to the Driver	88
	Generating Other Output File Types	90
	Creating Libraries	91
	Adding and Updating Files in Libraries	91
	Driver Options for Intermediate Forms of Output	91
	Output File Types	93
	Controlling Driver Information	95
	Using a Driver Options File	95
	Using Makefiles	97
	Green Hills Equivalents to GNU Tools	100
	Generating Dependency and Header File	
	Information	100
5	Developing for PowerPC	103
	PowerPC Characteristics	104
	Register Usage	105
	Data Type Sizes and Alignment Requirements	106
	Calling Conventions	107
	Structure Packing	108

Contents

Understanding Structure Packing	108
Using #pragma pack to Pack All Instances of a Structure Type	110
Using the Packing Builder Option to Pack All Structures	111
Pointing to Packed Structures with the __packed Type Qualifier	111
Specifying a PowerPC Target	113
PowerPC Processor Variants	114
Generating Debugging Information	115
Obtaining Profiling Information	117
Using Your Own Header Files and Libraries	122
Instructing the Compiler to Search for Your Headers	122
Instructing the Compiler to Link Against Your Libraries	125
Controlling the Assembler	125
Controlling the Linker	126
Text and Data Placement	127
Default Program Sections	127
Custom Program Sections	128
Assigning Program Sections to ROM and RAM	132
Storing Global Variables in Registers	134
Near and Far Function Calls	135
Special Data Area Optimizations	137
Customizing the Green Hills Run-Time Environment	146
Other Topics	147
Renaming the Output Executable	147
Specifying an Alternate Program Start Address	147
Writing Interrupt Routines	147
Symbolic Memory-Mapped I/O	148
Verifying Program Integrity	150
6 Builder and Driver Options	151
Target Options	154
Register Description File	154
Endianness	154
Floating-Point	154
Text and Data Placement	157
Instruction Set	161

Contents

Operating System	162
Project Options	172
Object File Output Directory	172
Source Root	172
Include Directories	172
Library Directories	173
Libraries	173
Output Filename	174
Source Directories Relative to This File	174
Intermediate Output Directory Relative to Top-Level Project	175
Optimization Options	176
Optimization Strategy	176
Intermodule Inlining	178
Automatic Vector Optimization	178
Linker Optimizations	178
Interprocedural Optimizations	179
Optimization Scope	179
Individual Functions	185
Debugging Options	187
Debugging Level	187
Profiling - Call Count	187
Profiling - Coverage	188
Profiling - Target-Based Timing	188
Run-Time Error Checks	188
Run-Time Memory Checks	191
Preprocessor Options	192
Define Preprocessor Symbol	192
Undefine Preprocessor Symbol	192
Display includes Preprocessor Directives Listing	192
C/C++ Compiler Options	194
C Language Dialect	194
C Japanese Automotive Extensions	195
C++ Language Dialect	195
C++ Libraries	196
C++ Exception Handling	196
Allow C++ Style Slash Comments in C	196
ANSI Aliasing Rules	196
MISRA C 2004	197

Contents

MISRA C 1998	216
Data Types	225
Alignment & Packing	226
C/C++ Data Allocation	227
Special Tokens	228
C++	229
Assembler Options	241
Source Listing Generation	241
Source Listing Generation Output Directory	241
Preprocess Assembly Files	241
Preprocess Special Assembly Files	242
Interleaved Source and Assembly	242
Additional Assembler Options	242
Assembler Command File	243
Support for C Type Information in Assembly	243
Linker Options	244
Output File Type	244
Generate Additional Output	244
Executable Stripping	245
Start Address Symbol	245
Append Comment Section with Link-Time	
Information	246
Preprocess Linker Directives Files	246
Linker Warnings	246
Raw Import Files	246
Linker Directive Files with Non-standard	
Extensions	247
Additional Linker Options (beginning of link	
line)	248
Additional Linker Options (before start file)	248
Additional Linker Options (among object files)	249
Linker Command File	249
Linker Optimizations	250
Start and End Files	252
Symbols	253
Linker Output Analysis	255
Link-Time Checking	257
Compiler Diagnostics Options	259
Warnings	259
Remarks	259

Contents

	Maximum Number of Errors to Display	259
	Redirect Error Output to File	260
	Quit Building if Warnings are Generated	260
	Display Version Information	260
	Varying Message Format	260
	C/C++ Messages	261
	DoubleCheck (C/C++) Options	266
	DoubleCheck Level	266
	DoubleCheck Report File	266
	DoubleCheck Config File	267
	DoubleCheck Errors to Ignore	267
	DoubleCheck Output that Stops Builds	267
	HTML Compiler Options	268
	Alternate Source Name	268
	HTML File Compression	268
	Alternate Compression Table	268
	Advanced Options	269
	Target Options	269
	Project Options	273
	Optimization Options	287
	Debugging Options	292
	Preprocessor Options	299
	C/C++ Compiler Options	303
	Assembler Options	309
	Linker Options	309
	Compiler Diagnostics Options	312
	Support Diagnostics Options	313
	HTML Compiler Options	314
7	Optimizing Your Programs	315
	Optimization Descriptions	316
	Inlining Optimizations	317
	Interprocedural Optimizations	330
	Loop Optimizations (including SIMD Vectorization)	339
	General Optimizations	351
	Default Optimizations	362
8	The DoubleCheck Source Analysis Tool	365
	Introduction	366

Contents

Enabling DoubleCheck	367
Specifying a DoubleCheck Report File	367
Using Custom Functions with DoubleCheck	368
Specifying Function Properties in a DoubleCheck Configuration File	369
Specifying Function Properties with Pragma Directives	370
Property Types	370
Viewing DoubleCheck Reports	373
Launching the Report Viewer from the MULTI Project Manager	373
Launching the Report Viewer From the Command Prompt	373
Controlling DoubleCheck Output	374
Promoting Source Analysis Errors and Warnings	374
Ignoring Source Analysis Errors and Warnings	375
Using DoubleCheck With Run-Time Error Checking	376
Source Analysis Error and Warning Messages	376
Error Messages	376
Warnings	386

Part II Using Advanced Tools

9 The asppc Assembler	391
Running the Assembler from the Builder or Driver	392
Generating Assembly Language Files	392
Running the Assembler Directly	393
Assembler Options	394
PowerPC-Specific Assembler Options	394
General Assembler Options	394
Assembler Syntax	396
Identifiers	396
Reserved Symbols	396
Source Statements	399
Expressions	401
Assignment Statements	401
Integral Expression Operators	401
Relocation Expression Operators	403

Contents

	C Type Information Operators	403
	Expression Types	406
	Type Combinations	406
	Labels	407
	Named Labels	407
	Temporary Labels	407
	Current Location	408
	-list File Output	409
	Reading the Compiler's Assembly Output	411
	Header Comment File	411
	Function Comment Section	412
	File Comment Section	412
	Source Line Comments	412
10	Assembler Directives	413
	Alignment Directives	415
	Conditional Assembly Directives	416
	Data Initialization Directives	417
	File Inclusion Directives	418
	Macro Definition Directives	418
	Repeat Block Directives	420
	Section Control Directives	421
	Source Listing Directives	424
	Symbol Definition Directives	426
	Symbolic Debugging Directives	427
	Miscellaneous Directives	428
11	The elxr Linker	431
	Running the Linker from the Project Manager or Driver	433
	Linker-specific Options	434
	Specifying the Program Entry Point	437
	Configuring the Linker with Linker Directives Files	437
	Linker Directives File Syntax	438
	Setting Linker Options with the OPTION Directive	438
	Setting Defaults with the DEFAULTS Directive	439
	Defining a Memory Map with the MEMORY Directive	440

Contents

Defining a Section Map with the SECTIONS Directive	442
Modifying your Section Map for Speed or Size	458
Customizing the Run-Time Environment Program Sections	459
Symbol Definitions	463
Beginning, End, and Size of Section Symbols	463
End of Function Symbols	464
Linker Generated Tables	465
Forcing The Linker to Pull In a Module From a Library	466
Deleting Unused Functions	467
Advanced Linker Features	469
Code Factoring	469
Run-Time Clear and Copy Tables	472
12 The ax Librarian	473
Creating a Library from the Project Manager	474
Creating a Library from the Compiler Driver	474
Modifying Libraries	475
Librarian Commands	476
Librarian Command Modifiers	477
Librarian Options	478
Examples	478
Creating and Updating a Table of Contents	478
13 Utility Programs	481
The gaddr2line Utility Program	484
The gasmlist Utility Program	485
The gbin2c Utility Program	487
Global Options	487
Module Inclusion Options	490
Local Options	490
The gbincmp Utility Program	496
The gbldconvert Utility Program	498
The gbuild Utility Program	499
Using gbuild	503
Implicit Dependency Analysis	505

Contents

The gcolor Utility Program	506
Color Configuration Files	507
Rules Files	509
The gcompare Utility Program	511
The gdump Utility Program	514
Debugging File Options	514
ELF File Options	516
COFF File Options	518
BSD File Options	519
The gfile Utility Program	521
The gfunsize Utility Program	522
The ghide Utility Program	523
Using ghide	523
The gmemfile Utility Program	524
Data Splitting	526
The gnm Utility Program	531
General Options	531
ELF File Options	531
BSD File Options	533
COFF File Options	534
Output Formats	534
The gpjmodify Utility Program	537
Editing Existing .gpj Files	537
Creating a New .gpj File	538
Generating a Makefile	539
The grun Utility Program	540
The gsize Utility Program	542
The gsrec Utility Program	544
S-Record Output Format	546
Data and Termination Records	547
Data Splitting	548
The gstack Utility Program	551
Caveats	552
The gstrip Utility Program	554
The gversion Utility Program	555
The gwhat Utility Program	557
Version Extract Mode	557

Contents

Version Update Mode	558
---------------------	-----

Part III Language Reference

14 Green Hills C	563
Specifying a C Language Dialect	564
ANSI C	565
ANSI C Extensions	565
Strict ANSI C	572
GNU C	573
GNU C Only Extensions	573
GNU C/C++ Extensions	576
Strict ISO C99	579
ISO C99	579
Preprocessor Support in C99 Mode	580
Enforced Requirements in C99 Mode	580
Different Features in C99 Mode	580
K&R C	582
K&R Mode Extensions to K&R C	583
Motor Industry Software Reliability Association (MISRA) Rules	587
Japanese Automotive C Extensions	587
Type Qualifiers	588
__bterversed	588
__bigendian and __littleendian	589
__packed	589
volatile	590
const	590
Thread-Local Storage for Cafe OS	590
Assignment and Comparisons on struct and union Types	591
Bitfields	592
ANSI C Limitations	592
Signedness of Bitfields	592
Size and Alignment of Bitfields	594
Enumerated Types	595
Functions with Variable Arguments	595
The <varargs.h> Facility	596

Contents

The <stdarg.h> Facility	597
The asm Statement	598
The Preprocessor	600
Preprocessor Output File	600
Extended Characters	600
Compiler Support for Multi-Byte Characters	601
Kanji Character Support	603
Compiler Limitations	604
ANSI C Implementation-Defined Features	605
Translation J.3.1	606
Environment J.3.2	607
Identifiers J.3.3	607
Characters J.3.4	608
Integers J.3.5	609
Floating-Point J.3.6	610
Arrays and Pointers J.3.7	610
Registers J.3.8	611
Structures, Unions, Enumerations and Bit-fields	
J.3.9	611
Qualifiers J.3.10	613
Declarators J.3.11	613
Statements J.3.12	613
Preprocessing Directives J.3.13	613
Library Functions J.3.14	614
Locale-Specific Behavior(J.4)	615
Additional Specifications	616
15 Green Hills C++	621
Specifying a C++ Language Dialect	622
Specifying C++ Libraries	623
Standard C++	624
Extensions Accepted in Normal C++ Mode	624
Embedded C++	627
Differences Between Standard C++ and Embedded	
C++	627
Extended Embedded C++	629
Features Supported by Each C++ Dialect	629
Standard C++ with ARM Extensions	630

Contents

GNU C++	631
GNU C++ Extensions	631
Template Instantiation	634
Prelinker Template Instantiation	635
Link-Once Template Instantiation	637
Instantiation Pragma Directives	638
Implicit Inclusion	640
Exported Templates	641
Multiple and Virtual Inheritance	643
Exception Handling	643
Namespace Support	645
Dependent Name Lookup	645
Lookup Using the Referencing Context	645
Argument Dependent Lookup	647
Precompiled Header Files	648
Manual PCH Processing	648
Automatic PCH Processing	649
Other Ways to Control PCHs	653
Performance Issues	653
Linkage	654
Post Processing in C++	655
The C++ decode Utility	656
C++ Implementation-Defined Features	656
Deprecated C++ Headers	661
16 Macros, Pragma Directives, and Intrinsics	663
Predefined Macro Names	664
Macro Formats	664
Macro Names Required by ANSI C and C++	664
Language Identifier Macros	666
Green Hills Toolchain Identification Macros	667
PowerPC-Specific Predefined Macro Names	668
Endianness Macros	669
Data Type Macros	669
Floating-Point Macros	670
MISRA Macros	672
Memory Model Macros	672

Contents

Alignment and Packing Macros	673
Current File and Function Macros	674
Object Format Macros	675
Optimization Predefined Macro Names	675
C++ Macros	675
Operating System-Specific Macros	676
Deprecated Macros	676
Pragma Directives	679
General Pragma Directives	680
Green Hills Extension Pragma Directives	683
Intrinsic Functions	692
Arithmetic Operation Instructions	697
Paired Single Extensions	697
Four Wide Vectors with Paired Singles	700
17 Libraries and Header Files	703
The Green Hills Header Files and Standard Libraries	705
C and C++ Header File Directories	705
Library Directories	705
Libraries	705
Automatic Searching of Libraries	708
Advanced Library Topics	709
Less Buffered I/O	709
64-bit Integer Arguments and printf	710
Memory Allocation and alloca()	711
Memory Allocation and malloc()	714
Using Thread-Safe Library Functions	714
Customizing Thread-Safe Library Functions	720
Green Hills Standard C Library Functions	721
Standard C Functions in libansi.a	722
libmath.a Functions	738
libind.a Functions	750
libstartup.a Functions	750
Customizing the Run-Time Environment Libraries and Object Modules	751
Creating a Project with a Customizable Run-Time Environment	751
Startup Module crt0.o	753
Low-Level Startup Library libstartup.a	755

Contents

Low-Level System Library libsys.a	756
Board Initialization Library libboardinit.a	760
Features that Depend on the Default Green Hills Startup Code	761
18 Mixing Languages and Writing Portable Code	763
Mixing Languages	764
Initialization of Libraries	765
Examples of main() Programs	765
Performing I/O on a Single File in Multiple Languages	766
C Routines and Header Files In C++	766
Using C++ in C Programs	767
Function Prototyping in C vs. C++	768
Mixing C and Assembly Language	769
Identifiers	769
Calling Subroutines	770
Accessing Variables	770
Writing Portable Code	774
Compatibility Between Green Hills Compilers	775
Word Size Differences	775
Endianness Problems	776
Alignment Requirements	777
Structures, Unions, Classes, and Bitfields	778
Assumptions About Function Calling Conventions	779
Pointer Issues	780
NULL Pointer	780
Character Set Dependencies	780
Floating Point Range and Accuracy	781
Operating System Dependencies	781
Assembly Language Interfaces	781
Evaluation Order	782
Machine-Specific Arithmetic	782
Illegal Assumptions About Compiler Optimizations	783
Memory Optimization Restrictions	784
Problems with Source Level Debuggers	785
Problems with Compiler Memory Size	785

Contents

19	Enhanced asm Macro Facility for C and C++	787
	Definition of Terms	789
	Using and Defining the asm Macro Facility	789
	Pseudo-Code Definition	790
	Use	790
	Using asm Macros	791
	Definition	792
	Storage Modes	793
	asm Body	794
	PowerPC asm Macros	795
	Guidelines for Writing asm Macros	798
20	The ELF File Format	801
	Formats of Relocatable and Executable Files	802
	32-Bit ELF Data Types	803
	ELF Headers	804
	ELF Identification	807
	ELF Sections	808
	Section Headers	808
	Symbol Tables	818
	Symbol Binding	819
	Symbol Type	819
	Symbol Values	820
	String Tables	821
	Program Headers	822
	Program Types	823
	Program Attribute Flags	824
Part IV	Appendices	
A	GNU Options	829
B	Green Hills Project File Format	831
	Green Hills Project File Syntax	832
	The Header Section	832
	The Project Options Section	833

Contents

	The Children Section	833
	Top Project Directives	835
	Group 1 Directives	836
	Group 2 Directives	836
	Conditional Control Statements	837
	Conditional Control Statement Syntax	837
	Special Conditional Control Statements	840
	Macro Functions	841
C	Customization Files	843
	Components of Customization Files	844
	Customization File Definitions	844
	Adding a Customization File to a Project	849
	Extended Examples	849
	Customization File Syntax	856
	FileType Syntax	856
	Context Sensitive Variables	858
D	Third-Party License and Copyright Information	865
	Index	867

Preface

This Preface Contains:

- About This Book
- The MULTI 5.3 Document Set
- Conventions Used in the MULTI Document Set

This section discusses the purpose of the manual, typographical conventions used, and the MULTI documentation set.

About This Book

This book provides comprehensive documentation of the components that make up the MULTI toolchain. It is divided into the following parts:

- *Part I: Using the MULTI Builder and Compiler Drivers* — Documents how to use the MULTI Builder or compiler drivers to control the toolchain. Includes an introduction to the Builder and compiler drivers, discussions of commonly-used features of the compiler for your target environment, and descriptions of all the Builder and driver options.
- *Part II: Using Advanced Tools* — Documents the other elements of the toolchain: the assembler, linker, librarian, and utility programs.
- *Part III: Language Reference* — Documents the Green Hills implementation of high level languages, including macros, `#pragma` directives, libraries, headers, and optimizations.
- *Part IV: Appendices* — Documents the use of GNU compiler options and advanced options for customizing various file types.



Note New or updated information may have become available while this book was in production. For additional material that was not available at press time, or for revisions that may have become necessary since this book was printed, please check your MULTI installation directory for release notes, **README** files, and other supplementary documentation.

The MULTI 5.3 Document Set

The primary documentation for using MULTI is provided in the following books:

- *MULTI: Getting Started* — Describes how to install MULTI, and takes you through creating, building, and debugging an example project.
- *MULTI: Licensing* — Describes how to obtain, install, and administer Green Hills licenses.
- *MULTI: Building Applications* — Describes how to use the MULTI Project Manager and compiler drivers and the tools that compile, assemble, and link your code. Also describes the Green Hills implementation of supported high-level languages.
- *MULTI: Configuring Connections* — Describes how to set up your target debugging interface for use with MULTI and how to configure connections to your target.
- *MULTI: Debugging* — Describes how to use the MULTI Debugger and its associated tools.
- *MULTI: Debugging Command Reference* — Describes how to use Debugger commands and provides a comprehensive reference of Debugger commands.
- *MULTI: Editing Files and Configuring the IDE* — Describes how to use the MULTI Editor and MULTI Launcher, how to use a version control system with MULTI, and how to configure the MULTI IDE.
- *MULTI: Scripting* — Describes how to create MULTI scripts and contains information about the MULTI-Python integration.

For a comprehensive list of the books provided with your MULTI installation, see the **toc_target.pdf** file located in the **manuals** subdirectory of your installation.

Most books are available in the following formats:

- A printed book (select books are not available in print).
- Online help, accessible from most MULTI windows via the **Help** → **Manuals** menu.
- An electronic PDF, available in the **manuals** subdirectory of your installation.

Conventions Used in the MULTI Document Set

All Green Hills documentation assumes that you have a working knowledge of your host operating system and its conventions, including its command line and graphical user interface (GUI) modes. For example, you should know how to use basic commands, how to open, save, and close files, and how to use a mouse and standard menus.

Green Hills documentation uses a variety of notational conventions to present information and describe procedures. These conventions are described below.

Convention	Indication	Example
bold type	Filename or pathname	C:\MyProjects
	Command	setup command
	Option	-G option
	Window title	The Breakpoints window
	Menu name or menu choice	The File menu
	Field name	Working Directory:
	Button name	The Browse button
<i>italic type</i>	Replaceable text	-o filename
	A new term	A task may be called a <i>process</i> or a <i>thread</i>
	A book title	<i>MULTI: Debugging</i>
monospace type	Text you should enter as presented	Type <code>help</code> <code>command_name</code>
	A word or words used in a command or example	The wait [-global] command blocks command processing, where -global blocks command processing for all MULTI processes.
	Source code	<code>int a = 3;</code>
	Input/output	<code>> print Test</code> <code>Test</code>
	A function	<code>GHS_System()</code>
ellipsis (...) (in command line instructions)	The preceding argument or option can be repeated zero or more times.	debugbutton [name]...

Convention	Indication	Example
greater than sign (>)	Represents a prompt. Your actual prompt may be a different symbol or string. The > prompt helps to distinguish input from output in examples of screen displays.	> print Test Test
pipe () (in command line instructions)	One (and only one) of the parameters or options separated by the pipe or pipes should be specified.	call <i>proc</i> <i>expr</i>
square brackets ([]) (in command line instructions)	Optional argument, command, option, and so on. You can either include or omit the enclosed elements. The square brackets should not appear in your actual command.	.macro <i>name</i> [<i>list</i>]

The following command description demonstrates the use of some of these typographical conventions.

gxyz [-*option*]... *filename*

The formatting of this command indicates that:

- The command **gxyz** should be entered as shown.
- The option -*option* should either be replaced with one or more appropriate options or be omitted.
- The word *filename* should be replaced with the actual filename of an appropriate file.

The square brackets and the ellipsis should not appear in the actual command you enter.

Introduction

Chapter 1

This Chapter Contains:

- The MULTI Integrated Development Environment
- The MULTI Launcher
- The MULTI Toolchain
- Accessing Online Help

This chapter provides an overview of the MULTI Integrated Development Environment, and then gives a brief introduction to the tools that are used to compile, assemble, and link code.

The MULTI Integrated Development Environment

MULTI is a complete Integrated Development Environment (IDE) designed especially for embedded systems engineers to assist them in compiling, optimizing, debugging, and editing embedded applications.

The MULTI IDE includes the following tools for each part of the software development process. Many of the MULTI tools can be launched from within the IDE or as separate stand-alone programs. Parenthetical text indicates corresponding executable files, which are located in your MULTI installation.

MULTI Launcher

MULTI Launcher (mstart) — The gateway to the MULTI IDE, which allows you to quickly launch any of the primary MULTI tools, access open windows, and manage MULTI workspaces.

Editing Tools

- **MULTI Editor (me)** — A graphical editor for modifying text files.
- **Checkout Browser (mcobrowse)** — A graphical viewer for files managed under a version control system.
- **Diff Viewer (diffview)** — A graphical viewer that displays differences between two text files.

Building Tools

- **MULTI Project Manager (mprojmgr)** — A graphical interface for managing and building projects.
- **Integrate (integrate)** — A graphical utility for configuring tasks, connections, and kernel objects across multiple AddressSpaces when using the INTEGRITY RTOS.

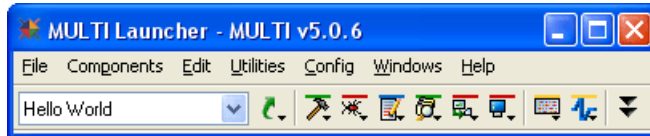
Debugging Tools

- **MULTI Debugger (multi)** — A graphical source-level debugger.
- **Instruction Set Simulator** — A tool that simulates your embedded processor on your development host.
- **EventAnalyzer (mevgui)** — A graphical viewer for monitoring the complex real-time interactions of an embedded RTOS such as INTEGRITY or u-velOSity.
- **ResourceAnalyzer (wperf)** — A graphical viewer for monitoring the CPU and memory usage of an embedded system running the INTEGRITY RTOS.
- **Serial Terminal (mterminal)** — A serial terminal emulator for connecting to serial ports on embedded devices.
- **DoubleCheck** — A static source code analysis tool for locating programming problems that lead to security and reliability failures in software.
- **MULTI TimeMachine Tool Suite** — A wide variety of trace analysis tools that dramatically extend MULTI's trace and debugging capabilities.

Administrative Tools

- **Bug Report (gbugrpt)** — A utility for providing system configuration and tool version information to the Green Hills support staff.
- **Green Hills Probe Administrator (gpadmin)** — A graphical interface for configuring and managing Green Hills Probes and SuperTrace Probes.
- **Graphical Utilities (wgutils)** — A collection of utilities for analyzing and performing various operations on object files, libraries, and executables produced with the Green Hills toolchain.
- **MULTI License Administrator (mlmadmin)** — A graphical utility for managing Green Hills tools licenses.

The MULTI Launcher



The MULTI Launcher (**mstart**) provides a convenient way to launch frequently used tools, to create new or access recently used files and projects, and to manage MULTI workspaces. All of the main MULTI components can be accessed using the following buttons:

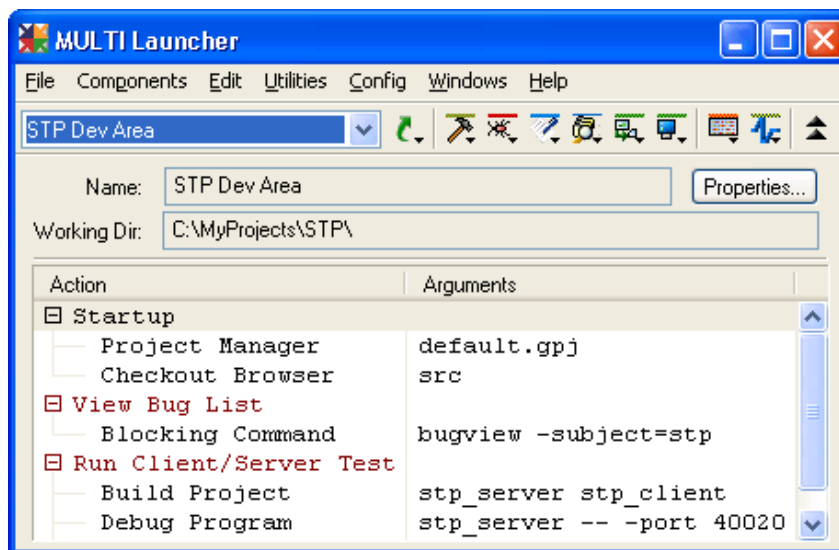
-  — Runs a shortcut or an action sequence in the current workspace. Also allows you to create a new workspace or create or edit a shortcut.
-  — Opens the Project Manager on a recent or new project.
-  — Opens the Debugger on a recent or new executable.
-  — Opens the Editor on a recent or new file.
-  — Opens the Checkout Browser on a recent or new checkout.
-  — Establishes a target connection or opens the Connection Chooser or Connection Organizer.
-  — Opens a Serial Terminal using a recent or new connection.
-  — Opens the EventAnalyzer (licensed separately).
-  — Opens the ResourceAnalyzer (licensed separately).
-  — Shows/hides the detail pane of the Launcher.

You can also launch the Green Hills License Administrator and (if installed) the Green Hills Probe Administrator from the **Utilities** menu.

During development, you can use the MULTI Launcher as a convenient, centralized window manager. You can access any of your open MULTI windows from the **Windows** menu of the Launcher.

MULTI Workspaces

The MULTI Launcher allows you to create and use *workspaces*. A MULTI workspace is a virtual area where the tools, files, and actions required for a particular project can be organized, accessed, and executed.



A workspace is typically created for each Top Project, and includes a working directory and a group of related *actions*—for example, opening a project in the MULTI Project Manager, connecting to a target, or performing a shell command. Actions are grouped into action sequences, so that a single mouse click can perform all the actions in the specified action sequence.

For more information, see Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.

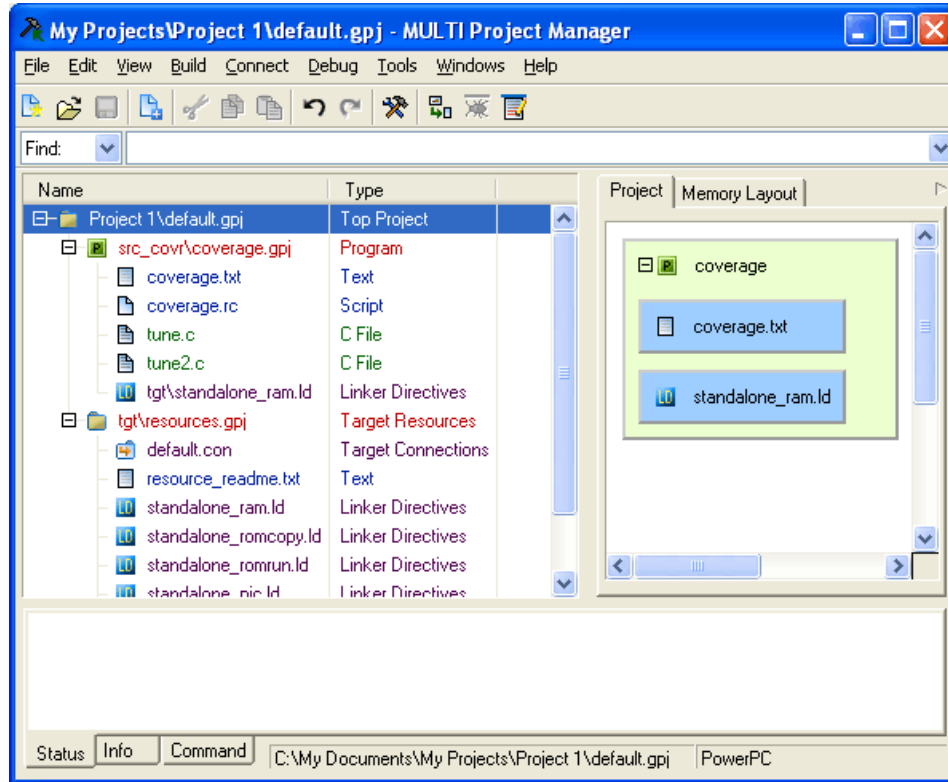
The MULTI Toolchain

The MULTI Project Manager

The MULTI Project Manager provides a convenient GUI interface for:

- Creating and organizing high-level and assembly language source files, object files, libraries, and supplementary files, and for setting file dependencies.

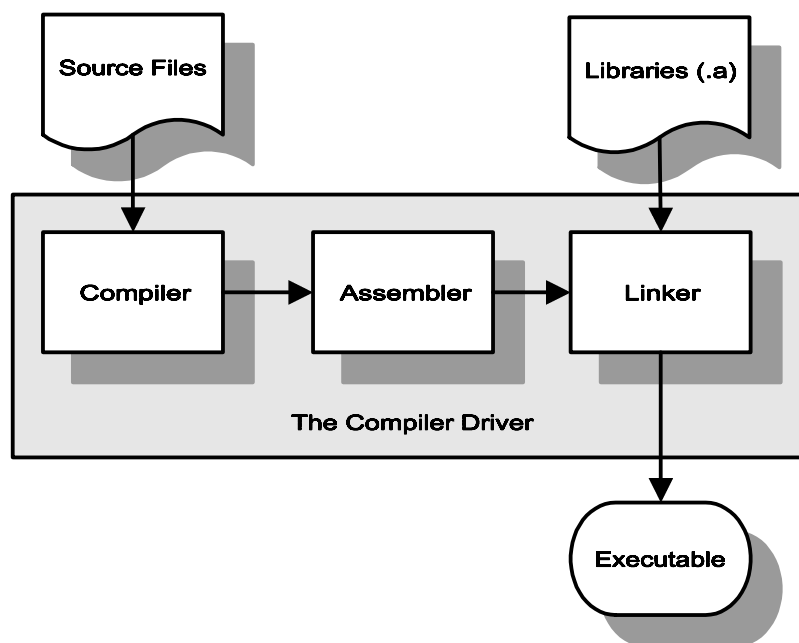
- Specifying build options to control how code will be processed by the toolchain.
- Invoking the Green Hills compilers, assembler, linker, and other utilities to build an executable.



To get started using the Project Manager, see Chapter 2, “The MULTI Project Manager”.

The C and C++ Compiler Driver

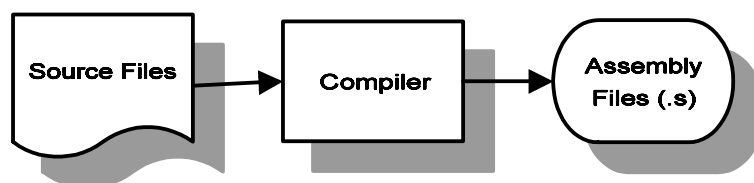
The compiler driver is the command line interface for invoking the components of the toolchain. It takes high-level source and other files as input, and invokes a compiler, the assembler, and the linker, to generate an executable.



To get started using the driver, see Chapter 4, “The Compiler Driver”.

The Optimizing Compilers

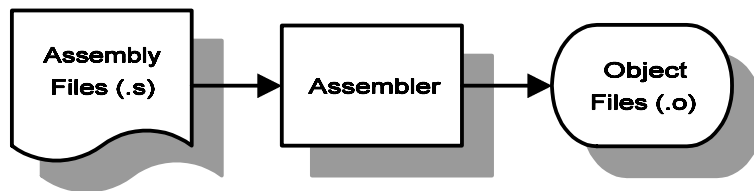
The Green Hills compilers are an integrated family of optimizing compilers that translate high-level language files into assembly language or object files.



There is a compiler for each of C and C++, which is composed of a language-specific front end, a global optimizer, and a target-specific code generator. Compatible subroutine calling conventions allow modules written in different languages to be combined. The compilers are invoked by the Builder or the appropriate compiler driver.

The asppc Assembler

The Green Hills assembler, **asppc**, translates PowerPC assembly language statements and directives into PowerPC machine code and data formats. The resulting file produced is an object file.



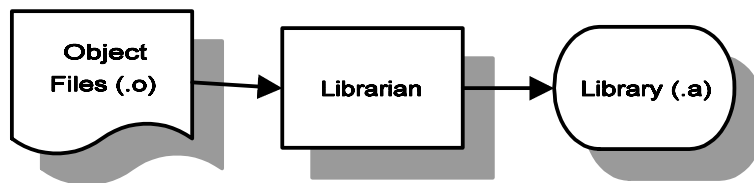
The assembler is ordinarily invoked by the Builder or compiler driver, when required, as part of the compilation process (see “Controlling the Assembler” on page 125). For information about invoking the assembler directly, and for a detailed description of its operations, see Chapter 9, “The asppc Assembler”.



Note The PowerPC compiler defaults to *binary code generation*, which allows the compiler to produce object files directly, without invoking the assembler, and results in reduced compilation times.

The ax Librarian

The Green Hills librarian, **ax**, combines object files into a library file.

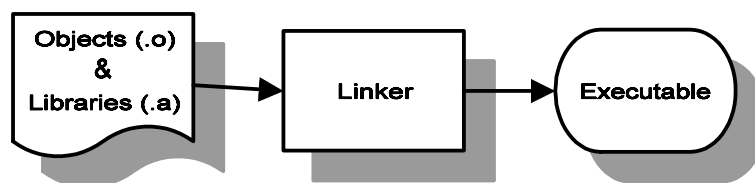


The linker can search libraries for object files to resolve external references. The linker pulls in an object file from a library only if it is referenced in the program.

The librarian is ordinarily invoked from the Builder or compiler driver (see “Creating Libraries” on page 91). For information about invoking the librarian directly, and for a detailed description of its operations, see Chapter 12, “The ax Librarian”.

The elxr Linker

The Green Hills linker, **elxr**, combines object files into an executable.



The linker resolves symbol references among all input files, and assigns symbols to memory locations.

The linker is ordinarily invoked by the Builder or compiler driver as part of the compilation process (see “Controlling the Linker” on page 126). For information about invoking the linker directly, and for a detailed description of its operations, see Chapter 11, “The elxr Linker”.

Optimized Libraries and Header Files

Green Hills provides optimized libraries and header files that support the standard libraries specified by each language. The compiler driver selects the appropriate libraries when linking, and header files when compiling. For more information, see Chapter 17, “Libraries and Header Files”.

Utility Programs

The Green Hills Utility Programs allow you to analyze and perform various operations on object files, libraries, and executables produced with the Green Hills toolchain. For more information, see Chapter 13, “Utility Programs”.

Accessing Online Help

Online help provides useful information about your MULTI tools and can be accessed in several different ways. The following sections describe how to access online help and how to use the Help Viewer.

Full Online Manuals

You can access an indexed, hypertext version of any MULTI manual by selecting it from the list that appears in the **Help** → **Manuals** menu. The MULTI Help Viewer opens on the manual specified.

You can also access manuals from the command line. Ensure that the MULTI installation directory is in your path and enter the following command:

```
helpview pathname | -m manual_name
```

where:

- *pathname* opens the **.chm** file specified in *pathname*. You must provide a full path.
- -m *manual_name* opens the manual *manual_name*. An example manual name is "MULTI: Debugging". If the manual name contains a space as in the preceding example, you must enclose it with quotation marks.

Context-Sensitive Help

Many MULTI windows and dialog boxes are linked to specific sections of the online manuals. To view the Help Viewer page that documents an active window or dialog box, press F1.

If you are using the MULTI Editor, you can also open context-sensitive help about a button or a menu item by selecting **Help** → **Identify** and then clicking the button or selecting the menu item.

Command, Configuration Option, and Keyword Help

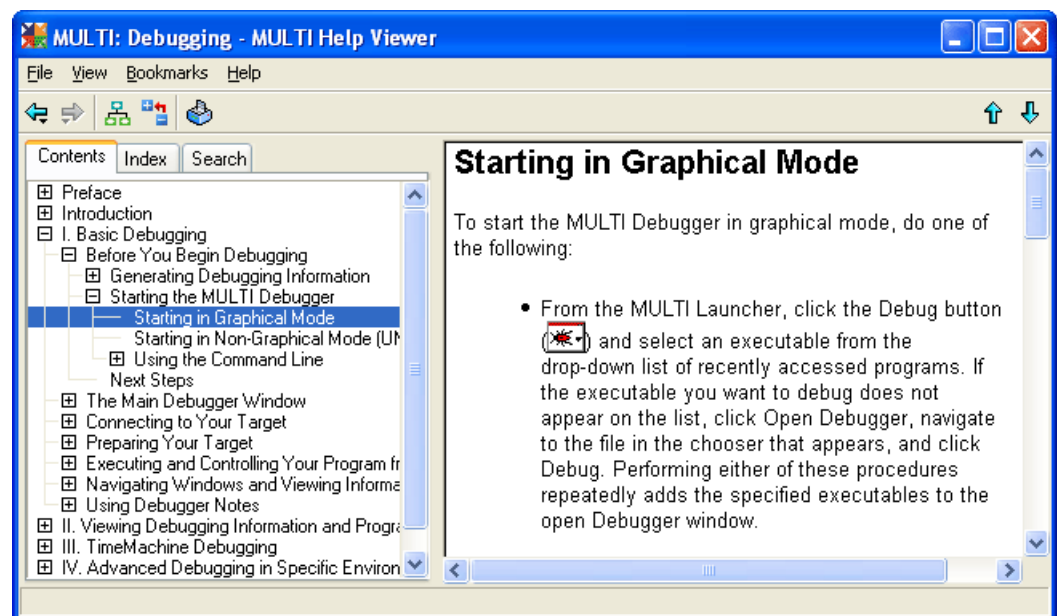
You can obtain help information about a MULTI Debugger command, a MULTI configuration option, or a keyword by typing `help command_name`, `help configuration_option`, or `help keyword` in the Debugger command pane. If

you specify the name of a command or configuration option, the Help Viewer opens on documentation for the command or option. If you specify a keyword (a word that is not a command or configuration option), the Help Viewer searches all manuals for *keyword*.

To print the basic syntax of a MULTI Debugger command to the command pane, type `usage command_name`.

The Help Viewer Window

The following graphic displays the Help Viewer window.



The Help Viewer toolbar contains the following buttons, from left to right:

- **Back** (↶) and **Forward** (↷) — Returns to pages you have visited during the current help session. Click the button once for each page you want to revisit. Click and hold the button to select from a list of pages you have visited. These buttons function across books.
- **View All Subtrees** (📁) — Toggles highlighting of the current selection's sub-sections (if any) in the **Contents** tab, and toggles the display of help for the current selection's sub-sections (if any) in the view pane. To disable highlighting in the **Contents** tab and to display help for only the selected item, click this button again. When you open the Help Viewer, this button defaults to its last value.

- **Contract All Unselected** () — Collapses hierarchies located in the **Contents** tab if the hierarchies do not contain any selections.
- **Print** () — Prints the help page displayed in the view pane. To create a help page consisting of mixed topics, hold down the Ctrl key while clicking separate topics. MULTI combines the separate topics and orders them by their location in the **Contents** tab.
- **Previous Page** () and **Next Page** () (located at far right) — Displays the previous or following section as ordered in the **Contents** tab. If the previous or next section includes sub-sections and the **View All Subtrees** button is enabled, the Help Viewer displays the sub-sections (if any) along with the section.

The MULTI Help Viewer contains two panes. The right-hand pane, or the *view pane*, displays the help page for your selection. The left-hand pane contains the following three tabs:

- **Contents** — Displays the parts, chapters, sections, and sub-sections for the manual you are viewing. Click a plus or minus icon to show or hide headings for embedded sections. Click a heading to display the associated help page in the view pane.
- **Index** — Displays index entries for the manual. To search the index entries, enter a word or words in the text box located at the top of the pane. MULTI returns results that contain the word(s) you typed. Click an index entry to display the corresponding help page in the view pane.
- **Search** — Provides an interface that allows you to search for specific words in the current manual or in all manuals.

Navigating Manuals in the Help Viewer

Navigating manuals in the MULTI Help Viewer is easy. This section describes different methods of finding information.

Viewing Previous and Next Sections

To display the previous or following section as ordered under the **Contents** tab, do one of the following:

- Click the  or  button located in the upper-right corner of the Help Viewer window.
- Select **View → Previous Contents Entry** or **View → Next Contents Entry**.

If the previous or next section includes sub-sections and the **View All Subtrees** button () is enabled, the Help Viewer displays the sub-sections (if any) along with the section.

Returning to Previously Viewed Help Pages

To return to pages you have visited during the current help session, do one of the following:

- Click the  or  button located in the left corner of the toolbar. Click the button once for each page you want to revisit. Click and hold the button to select from a complete list of pages you have visited during the current help session. These buttons function across books.
- Select **View → Back** or **View → Forward** from the menu bar. This feature functions across books.

Linking to Related Topics

The underlined links available in the view pane allow you to jump to pages related to the topic you are viewing.

If, when you click a link, a dialog box appears with the message:

You have requested information that the Help Viewer is unable to retrieve.

one of the following factors may be the cause:

- The manual that the link points to may not be included with your distribution. For a list of available manuals, look in the manuals directory (or directories) of your Green Hills Software distribution(s).
- If the link points to an INTEGRITY or u-velOSity manual, you may be using a version of the operating system that does not support links from MULTI help.
- If the link points to an INTEGRITY or u-velOSity manual, you may need to set the location of the installed OS distribution so that the Help Viewer can locate the relevant help files. For information about how to do this, see “Using MULTI with INTEGRITY or u-velOSity” in Chapter 2, “Installation” in the *MULTI: Getting Started* book.



Note If you are using MULTI with multiple target architectures (such as ARM and PowerPC), tools of the same version number should be installed into a single directory, regardless of differing architectures. Note, however, that links to target-specific books may yield unpredictable results in this situation.

Bookmarking Help Pages

You can use bookmarks to return to pages you have visited across MULTI sessions. To bookmark a page, select **Bookmarks** → **Bookmark This Page**.

You can bookmark a help page consisting of mixed topics by holding down the Ctrl key while clicking separate topics. MULTI combines the separate topics in the view pane and orders them by their location under the **Contents** tab. Bookmark the page as usual.

The **Bookmarks** menu lists all your bookmarks from across MULTI manuals and sessions. To return to a bookmarked page, do one of the following:

- Select **Bookmarks** and choose a bookmark.
- Select **Bookmarks** → **Manage Bookmarks**, click a bookmark, and click the **Show** button (🔍).
- From any MULTI tool, select **Help** → **Bookmarks** and choose a bookmark.

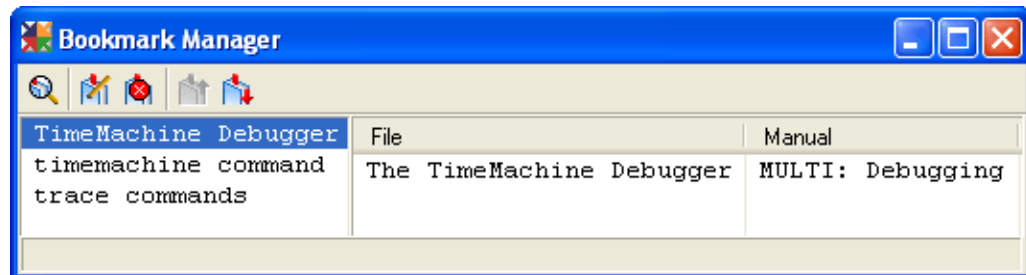
To rename, delete, or reorder existing bookmarks, launch the **Bookmark Manager** as described in the next section, “Managing Bookmarks” on page 15.

Managing Bookmarks

You can use the **Bookmark Manager** to rename and delete bookmarks, navigate to bookmarks, and modify the ordering of bookmarks in the **Bookmarks** menu.

To open the **Bookmark Manager**, do one of the following:

- In the Help Viewer, select **Bookmarks** → **Manage Bookmarks**.
- From any MULTI tool, select **Help** → **Bookmarks** → **Manage Bookmarks**.



The list on the left side of the **Bookmark Manager** displays the names of all the bookmarks that you have created. When a bookmark is selected, the file(s) and the manual marked by that bookmark are displayed in the right side of the window. The bookmark may reference multiple files (help pages) if, for example, you created the bookmark with multiple entries selected in the **Contents** tab or if you selected an index entry that pertained to multiple files.

The toolbar buttons in the **Bookmark Manager** allow you to perform the following operations on bookmarks:

- — Display the selected bookmark(s) in the Help Viewer.
- — Rename the selected bookmark.
- — Delete the selected bookmark(s).
- and — Move the selected bookmark up or down in the list. The order of the bookmarks in the **Bookmarks** menu mirrors the order of the bookmarks in the **Bookmark Manager**.



Note The and buttons are available for use on multiple bookmarks. You can select multiple bookmarks from the left side of the window by dragging to select rows or by using Shift or Ctrl to select items.

Opening Additional Manuals from the Help Viewer

To open a new manual in the current window, do one of the following:

- Select **File** → **Open Manual** and choose from the list of installed manuals.
- Select **File** → **Open Manual File** and navigate to a particular file in the file chooser that appears.

To open a new manual in a new window, select **View** → **Open Other Manuals in New Window** and then open the manual as usual. When you open the Help Viewer, the **Open Other Manuals in New Window** toggle option defaults to its last value.

Using the MULTI Builder and Compiler Drivers

Part I

The MULTI Project Manager

Chapter 2

This Chapter Contains:

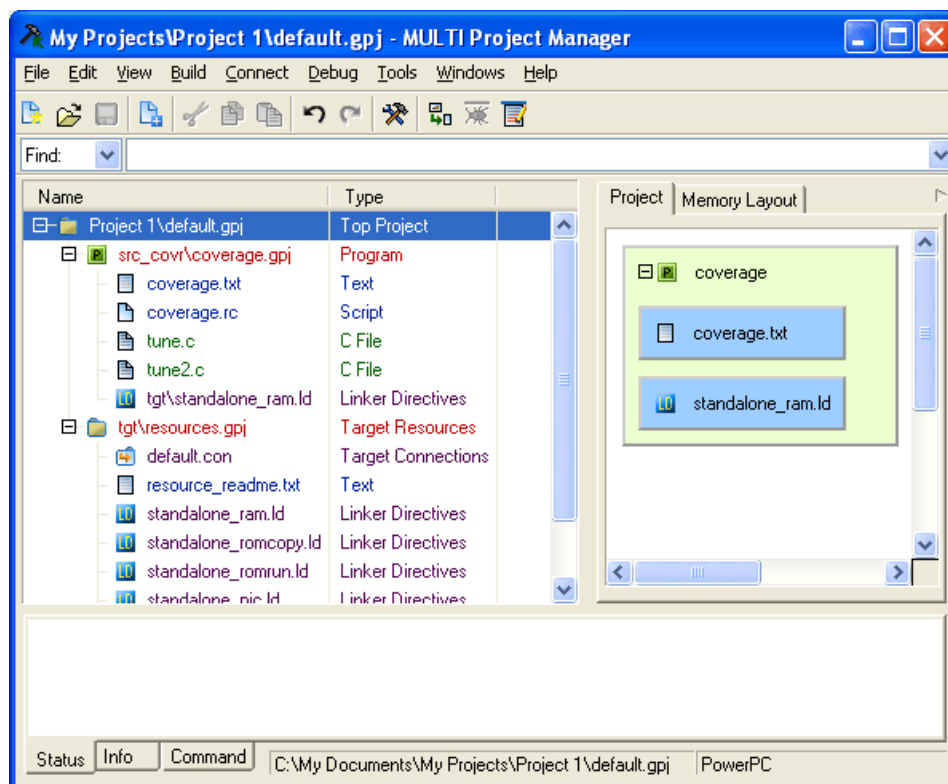
- Starting the MULTI Project Manager
- Creating A New Project
- Managing Your Project
- Building Your Project
- Setting Builder Options
- Working with Linker Directives Files

The MULTI Project Manager is a graphical tool that organizes your source and other input files, and controls the tools needed to compile your software project.

The term *project* is used to encompass all of the files that are used to build your application. Projects are defined in Green Hills *project files* (**.gpj**), which are similar to makefiles. The Project Manager maintains file dependencies, and sets the options used in building. Projects are organized in a tree structure, where the root of the tree is a Top Project (usually called **default.gpj**). For more information about Top Projects, see “Top Projects” on page 28.

To start the Project Manager:

- From the MULTI Launcher, click the **Launch Project Manager** button (🔧) and select either an existing Top Project or **Create Project**.



The primary components of the main Project Manager window are:

- *Title Bar* — Displays the location of the current Top Project. In the preceding screen, the project is **default.gpj**, which is the default project name for projects created by the **Project Wizard**.

- *Menu Bar* — For a complete list of all the menu items, see Chapter 3, “The Project Manager GUI Reference”.
- *Toolbar* — The buttons on the Project Manager toolbar are documented along with their menu equivalents in Chapter 3, “The Project Manager GUI Reference”.
- *File Shortcut Bar* — Allows you to quickly locate or build files and projects. For more information, see “Using the File Shortcut Bar” on page 43.
- *Project Tree* — Displays the files and other configurable items in the current project. The Project Manager displays the name and type for each item in the tree. You can also configure it to display the size and modified Builder options for applicable items. Projects (**.gpj**) and some configurable items have a small plus or minus sign to their left that allows you to expand or collapse their contents.
- *Status Tab* — Displays information about the current status of the build process.
- *Info Tab* — Displays information about the selected items.
- *Command Tab* — Displays the command that will be run to build the selected file. This shows which drivers options will be used to build the file.
- *Status Bar* — Displays the full path of the currently selected file. If a build is running, the status bar shows the progress of the build.
- *Target Indicator* — Displays the currently selected build target architecture and operating system.

Additionally, you can enable the *Advanced Views* pane to display the following tabs:

- *Project* — Displays the important components in your project, by using a layout that is similar to what you might draw on a white board.
- *Memory Layout* — Displays an overview of the section maps used in your project file. The information in this tab is only displayed after you have built the selected project. To learn about this functionality, see “Navigating the Memory Layout Tab” on page 45.
- *Integrate* — Displays a lightweight version of the Integrate Utility. This tab is only visible in INTEGRITY projects. For more information, see the *Integrate User’s Guide*.



Note To enable the **Advanced Views** pane, select **View** → **Show All Views**, or click the arrow on the right edge of the window, just below the **File Shortcut Bar**.

Starting the MULTI Project Manager

You can open the Project Manager from the MULTI Launcher, the MULTI Debugger, the command line, or a Windows shortcut. The Project Manager can only open Top Project files (usually called **default.gpj**). For more information about these files, see “Top Projects” on page 28. There are several ways to open the Project Manager:

- From the MULTI Launcher, click the **Launch Project Manager** button () and select either an existing project or **Create Project**.
- From the MULTI Launcher, select **Components** → **Open Project Manager** to open a file chooser where you can select your program’s Top Project.
- From the MULTI Debugger, select the **Tools** → **Project Manager** menu item.
- From the MULTI Debugger, enter the **builder** command on the Debugger command line.
- If you want to open MULTI from the command line, we recommended that you add the MULTI install directory to your path for easy access. After updating your path, enter the following command:

```
multi filename.gpj
```

where *filename.gpj* is your program’s Top Project. If you do not specify *filename.gpj*, the Launcher opens instead.

- On Windows, drag and drop a Top Project file to a MULTI icon to open it.

Creating A New Project

Before you begin development, you must create a Top Project with the **Project Wizard**. The Top Project will contain all of your code and defines any executables you plan on building. The **Project Wizard** provides a simple interface that allows you to:

- Set the name of the Top Project.

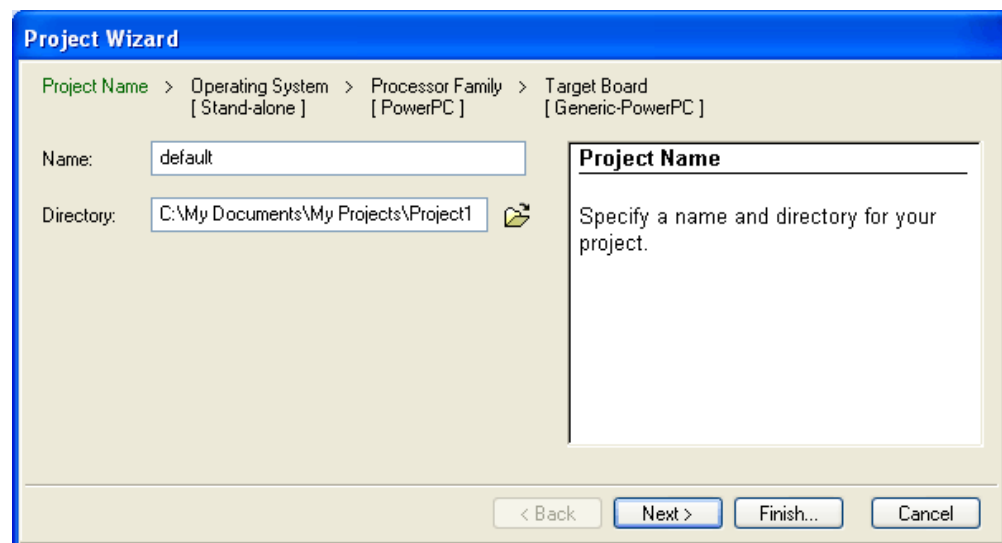
- Specify the directory where the Top Project will be created.
- Set the operating system used in your project.
- Specify your target board or processor.

When you create a new Top Project, the **Project Wizard** keeps track of your selections, and offers them as defaults when you create subsequent **Top Projects**. After the **Project Wizard** finishes creating your project, you will be prompted to add examples, executables, or other items appropriate to the target you selected.

If you are creating a Top Project for INTEGRITY, see the “Building INTEGRITY Applications” chapter in the *INTEGRITY Development Guide*.

Using the Project Wizard

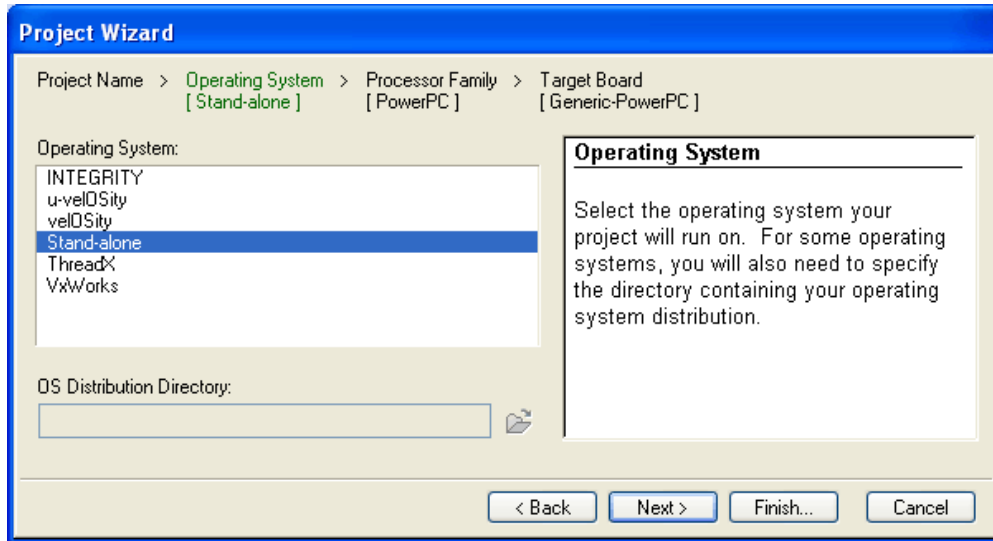
To open the **Project Wizard** and create a new Top Project, click the  button in the MULTI Launcher and select **Create Project**. The following screen appears:



On this screen, you can specify:

- The name of the top project and the directory where the project will be created. The directory you select should be empty or non-existent.

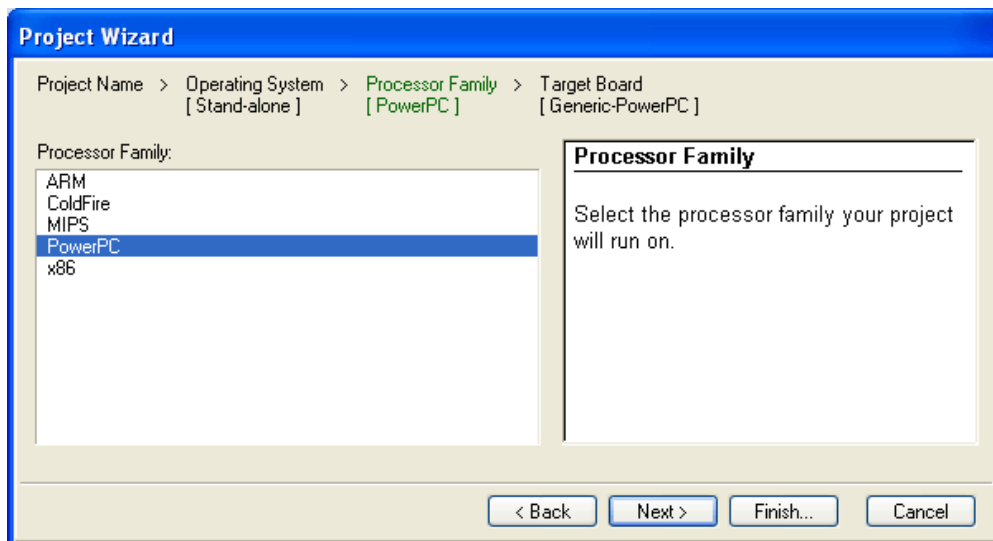
Click **Next** and the following screen appears:



On this screen, you can specify:

- The operating system running on your target. This book focuses on development for *stand-alone* targets, those which do not have a Real-Time Operating System (RTOS) running on them.

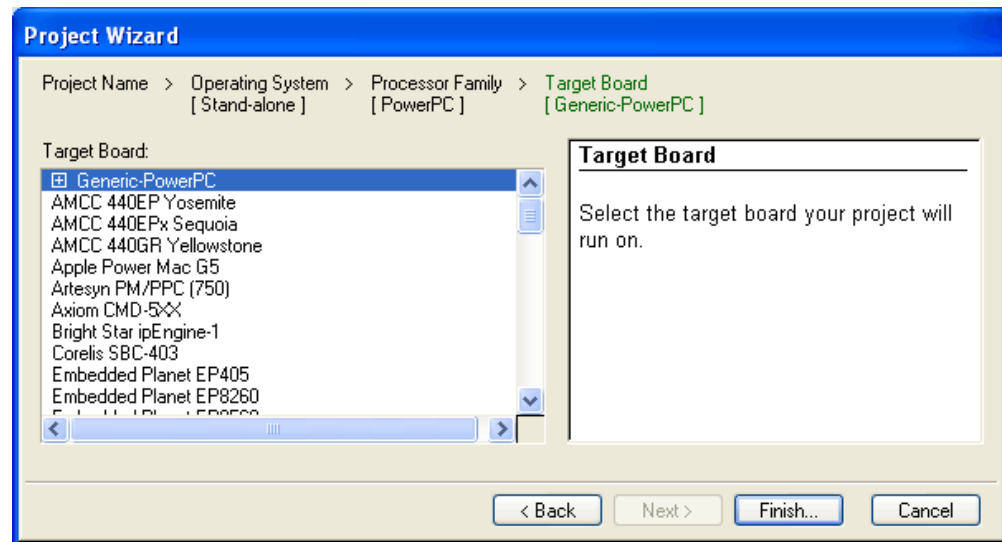
Click **Next**. If you have only installed one processor family for MULTI, the wizard selects that family and continues to the next step. Otherwise, it displays the following screen:



On this screen, you can specify:

- The processor family to which your target belongs.

Click **Next** and the following screen appears:



On this screen, you can specify:

- Your target board. If your target board is not listed or you are using custom hardware, click the plus sign to the left of the **Generic** item at the top of the list and select your processor. In this case, you will probably need to customize the board setup script and linker directives files the wizard creates for your target.

Click **Finish** to complete the setup of the Top Project. The **Project Wizard** creates a starting point for you to begin defining the structure of your project. The **Project Wizard** also generates files containing information specific to the target you selected. You can find these files in the Target Resources project created by the **Project Wizard** (see “Target Resources Project” on page 28).

A dialog box will open that informs you about the newly created framework. To disable this message in the future, select the check box **Never show this message again**. Click **OK** to continue.



Note If only one operating system or processor family is available, the **Project Wizard** will automatically select it and continue to the next screen.

Adding Executables and Examples to the Top Project

After you have created a Top Project, you are immediately given the opportunity to add executables and examples to your new Top Project.

To add new items to your project:

1. Select an executable or an example.
2. Click **Finish** to accept the default settings. For more information about these settings and how to modify them, see “Adding New Items to Your Project” on page 29.

Opening the Program in the MULTI Debugger

After you have completed the project setup, you can build your new application, connect to the Green Hills PowerPC simulator, and open it in the Debugger:

- Click the **Build** button () to build your project. You can follow the progress of the build in the **Status** tab at the bottom of the main Project Manager window.
- Click the **Connect** button () to open the **Connection Chooser**. The **Project Wizard** has created a connection called:

Target Board with Simulator for PowerPC

which is selected. Click **Connect** to use this connection.

- Click the **Debug** button () to open the application in the Debugger (if the **Debug** button is not already enabled, select a Program project to enable it).



Note For more information see:

- “Building Your Project” on page 42.
- Chapter 4, “Connecting to Your Target” in the *MULTI: Debugging* book.
- Chapter 6, “Executing and Controlling Your Program from the Debugger” in the *MULTI: Debugging* book.

Managing Your Project

After you have created an example project and are comfortable with the Project Manager, you can begin adding your own files to the skeleton project structure generated by the **Project Wizard**. The following provides information about the contents of the project structure, how to manage your projects, in addition to the various file formats used by Green Hills tools.

The **Project Tree** pane of the Project Manager provides a graphical representation of the hierarchy of your project files. It contains four columns:

- **Name** — The name of each file and its position in the project hierarchy.
- **Type** — The type of file, which is usually determined from its extension.
- **Options** — The compiler options that are set at this level of the hierarchy.
- **Size** — The amount of memory required for the file. Files must be built for this data to appear. If you hover your mouse over a value in this column, a tooltip containing the breakdown of memory needed for read-only and read-write sections appears.

By default, the **Options** and **Size** columns are not visible. To display these columns, enable **View** → **Show Options Column**, or **View** → **Show Size Column**.

The **Project Tree** pane displayed in the following shows the files generated by the **Project Wizard** for the example project **Hello World**:

Project1\Project1.gpj	Top Project
src\hello.gpj	Example
hello.c	C File
tgt\standalone_ram.ld	Linker Directives
tgt\resources.gpj	Target Resources
default.con	Target Connections
standalone_ram.ld	Linker Directives
standalone_romcopy.ld	Linker Directives
standalone_romrun.ld	Linker Directives
standalone_pic.ld	Linker Directives
standalone_pid.ld	Linker Directives
standalone_picpid.ld	Linker Directives

Files inherit compiler options from parent project files. For example, **src\hello.gpj** is a child that inherits options from its parent, **Project1\Project1.gpj**. **hello.c** inherits options from **src\hello.gpj** and from

Project1\Project1.gpj. For information about setting options, see “Setting Builder Options” on page 49.

Top Projects

The Top Project is the root of your project hierarchy, and the only type of project file that you can open with the Project Manager. Usually, Top Projects are named **default.gpj**. The Top Project encompasses all of your projects and defines your debugging environment. Among other things, it specifies:

- Your target architecture
- The operating system you are building for
- Customization files you are using, if any
- Build macro definitions
- Other projects (such as programs and libraries)
- Builder options inherited by other projects
- The target resources project (see “Target Resources Project” on page 28)

For information about the Top Project file format and advanced features, see Appendix B.

Target Resources Project

The target resources project (**tgt\resources.gpj**) is generated by the **Project Wizard** when you create a new Top Project. The files included in the target resources project contain information specific to the operating system and target board used for your project. The following is a non-exhaustive list of files that may be included in your Target Resources project:

- Target Connection File (**default.con**) containing Target Connection Methods that you can use to connect to your target board.
- BSP Description File (**default.bsp**) containing memory configuration information used by the **Integrate** utility. This file is only included in INTEGRITY projects.
- Linker Directives Files (**.ld**) specifying pre-configured memory layouts for your programs.

- Target Setup Scripts (**.mbs** and **.dbs**) containing commands required to initialize your board for running programs in RAM.
- Board-Specific Notes (**.txt** and **.notes**).
- Custom Library Source Code (**lib*.gpj**) containing customizable copies of the Green Hills startup, system, and board initialization libraries. These files are only added if you select to customize the library code in one or more of your programs.

All of these files are located in the **tgt** subdirectory of your project.

By default, all executables you create within your Top Project share one target resources project. This allows you to make modifications in one place (for example, altering a Linker Directives File), and to see the corresponding changes in all of your executables.

If you change the target for your Top Project (**Edit** → **Set Build Target**), a new target resources project is generated with information specific to your new target. The original Target Resources project and all files in the **tgt** directory are moved to **tgt.old**.

If necessary, you can copy files from the target resources to an individual executable using **Copy selected file Local** (see “The Edit Menu” on page 67).



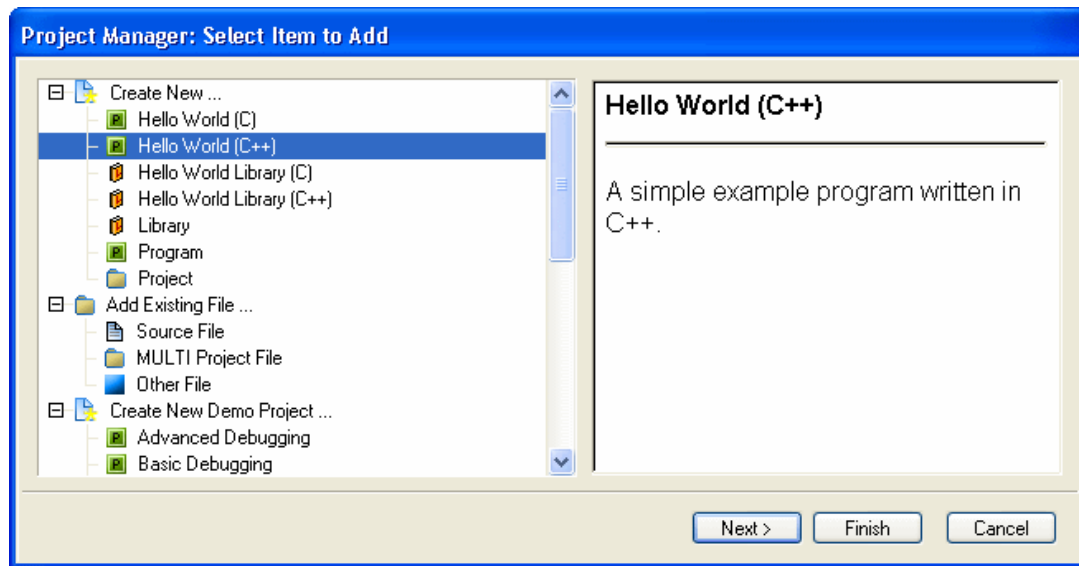
Caution After making local copies of customized library files, you might need to modify your program’s build options to make it use the local library instead of the shared one.

Adding New Items to Your Project

The Project Manager provides a **Select Item to Add** dialog box for adding new items to your project. Items that you might want to add to your project include Libraries, Projects for organizing your code, Programs to create an executable, or source files. This dialog opens after you create a Top Project with the **Project Wizard**. If you want to add an item to an existing project:

1. Select the item that you want to add a child to.
2. Click .

The contents of this dialog box vary based on the type of item you are adding a child to. For example, if you add a child to a stand-alone Top Project, it should look similar to the following image:



To add a new item:

1. Select the type of item you want to add.

If you are adding items to an INTEGRITY project, see the “Building INTEGRITY Applications” chapter in the *INTEGRITY Development Guide*.

2. Click **Finish** to accept the default settings for the item, or click **Next** to modify the settings.

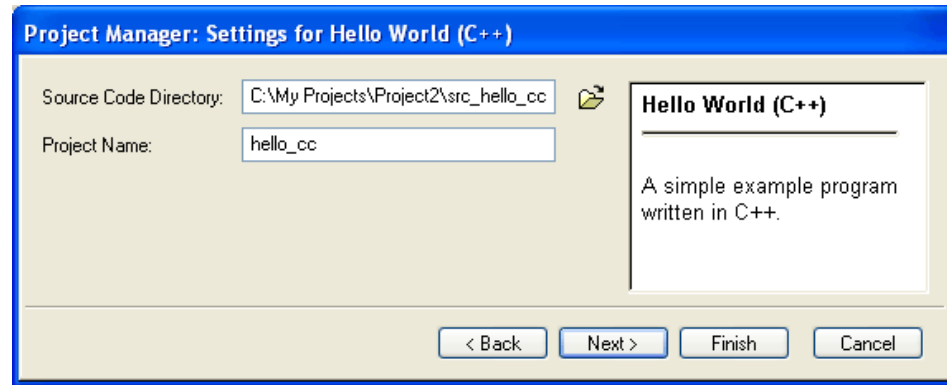
Each type of item has different settings available. The following sections describe several common settings. Documentation for other settings is shown in the text pane on the right side of the dialog box.

After you click **Finish**, the Project Manager generates any required files, and modifies the project hierarchy to include the new item. These changes are saved immediately.



Note When adding a source file, you will be presented with a file chooser to enter the name of the source file. If the file you enter does not exist, it will be added to the project hierarchy, but the file will not be created. To create the file, double-click the file to open it in the MULTI Editor, edit, and save the file.

Settings for your Project

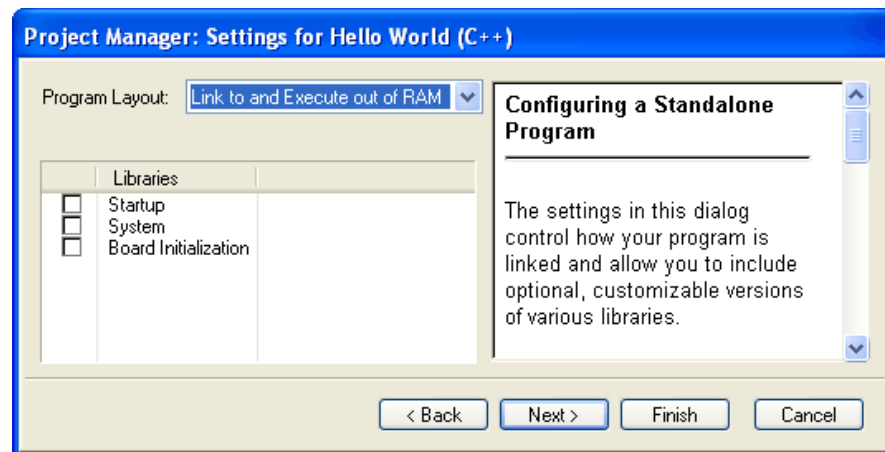


On this screen, you can specify:

- The directory to create the new executable or example.
- The name of the project file. For program projects, this is also the default name of the output executable.

Enter the specific settings for the selected item and click **Next**.

Settings for Stand-Alone Programs



On this screen you can specify:

- A standard program layout. The different layouts correspond to different linker directives (**.ld**) files. Each layout provides an appropriate section map for your program, and memory map of your target board. The default layout links to and executes out of RAM.

The layout that you choose is the one that will be used during linking. The Project Manager provides a number of linker directives files (in the Target Resources project) so that you can change the layout conveniently at later stages of development. For more information, see “Working with Linker Directives Files” on page 63.



Note Selecting **Link to and Execute out of ROM** or **Link to ROM and Execute out of RAM** might require you to include the Board Initialization Library in your program, or include other code that handles the target from reset and initializes the memory subsystems. See “Board Initialization Library libboardinit.a” on page 760 for a description of this interface and the **board_info.txt** file in your project, if one exists.

- Whether to use the standard pre-built startup code, or to customize it. The default is to use the pre-built code. To customize startup code, click the **Startup** box. If you choose to customize, the Project Manager adds an additional project file to your project that contains the source code to the Green Hills startup library, **libstartup.a** and **crt0.o**.
- Whether to use the standard pre-built low-level system code, or to customize it. The default is to use the pre-built code. To customize system code, click the **System** box. If you choose to customize, the Project Manager adds an additional project file to your project that contains the source code to the Green Hills system library, **libsys.a**.
- (Selected target boards only) Whether to include a board initialization library. The default is to omit the library. If this library is available, a check box for **Board Initialization** will appear in the list. This library provides basic memory and peripheral initialization code that allows the project to be built for ROM or flash. The library might also provide serial port code that allows standard I/O functions to use the serial port. If you choose to include this library, the Project Manager adds an additional project file that contains the source code to the Green Hills board startup library, **libboardinit.a**. The source code to the Green Hills system library **libsys.a** is also added because some files in the board initialization library reference that code.

The board initialization library is intended for use with hardware connections. If you include this library, you may not be able to run your program on a simulator.

- Other settings might appear on this screen in addition to those above. See the in-dialog documentation for more information about these settings.

The Project Manager adds files for customized libraries and startup code to the Target Resources project in your Top Project. By default, all programs under one Top Project share the same customized source code. For more information, see “Target Resources Project” on page 28.

Click **Finish** to complete adding new items to your project framework. If you decide that you want to change one of these settings later, you can change most of them using the **Configure** menu item (see “Configuring Existing Items” on page 33).

Adding Existing Files To Your Project

To add existing source and project files to a project:

1. Select the location within the hierarchy where you want to add the file.
2. Click **Edit** → **Add File into** *proj.gpj*, or right-click the project file and select **Add File into** *proj.gpj*.

Depending on what you have selected, either **Add File into** *Project* or **Add File after** *Project* is displayed.

3. In the file chooser, browse to the file and click **Add**.

For Windows

You can quickly add existing files to your project hierarchy by dragging the files from Windows Explorer into the Project Manager’s Project Tree.

For Linux/Solaris

To add multiple files quickly, you can specify a file pattern using the wildcard characters asterisk (*) and question mark (?) in the file chooser.

Configuring Existing Items

Many of the settings in the **Project Wizard** can be modified after you have created your project. To modify these settings:

1. Select the item you want to reconfigure.
2. From the toolbar select **Edit** → **Configure** or right-click your project file and select **Configure**.

The **Project Wizard** opens with a dialog box that indicates your project's current settings. For more information about the **Project Wizard** settings dialog boxes, see “Adding New Items to Your Project” on page 29.

Editing Project Files

You can edit any of the files contained within a project in the MULTI Editor by selecting it and pressing Ctrl + E, or right-clicking it and selecting **Edit** from the context-sensitive menu. You can also open source files and text files in the Editor by double-clicking their name.

You can access graphical interfaces to assist you in editing certain file types, by right-clicking them and selecting **Graphically Edit**:

- Double-click a **Target Connections** file (.con) to open the **Connection Organizer**, which allows you to edit or add a way of connecting to your target. For more information, see Chapter 4, “Connecting to Your Target” in the *MULTI: Debugging* book.
- **For Windows** — Files with registered file extensions can be opened with their open action.
- **For Linux/Solaris** — Several standard third-party file types have been added including .pdf. You must have Adobe Acrobat Reader installed, and the **acroread** executable must be in your path to view .pdf files from the Project Manager.

Moving Items in the Project Hierarchy

The files at each level of your project’s hierarchy are displayed in the order that you added them to the Project Manager. This is also the order in which the files are built. You can rearrange the order in which the files are displayed to change the compilation order or to help visualize your hierarchy.

To reorder files inside a single project file:

1. Select the file you want to move.
2. Press Ctrl+UpArrow to move the file up the hierarchy. Press Ctrl+DownArrow to move the file down the hierarchy.

To move files within a single project file or from one project to another, you can cut and paste files using either of the following methods:

- Ctrl + X and Ctrl + V, or
- the Cut () and Paste () buttons. The paste button is equivalent to the **Paste file as Link** menu items.

To paste files, select the file immediately above where you want the files to be added and click the Paste () button.



Note You can select multiple files to cut as long as they are all contained within the same project.

You can also copy and paste items as links or local items. **Copy file as Link** allows you to copy the selected items to the clipboard. **Paste file as Link** takes the clipboard items and creates a reference to them in the selected context. This does not create another copy of the items, so any changes made to the item will be reflected in all contexts that are linked.

Copy file Local copies the selected items to the clipboard. **Paste file Local** takes the clipboard items and makes a recursive copy of each item within the selected context. You will have your own distinct copy of the item, so subsequent changes made in the old copy will not be reflected in your new copy.

Common Scenarios

The following section provides you with additional information about various tools that you can use in the Project Manager.

Adding Header Files

You can add header files to your project hierarchy to quickly access them for editing. However, adding a header file to the hierarchy does not affect how the compiler locates the header file when it is compiling your source files.

To add a header file in your project hierarchy, add the file in the same manner as adding a source file (see “Adding Existing Files To Your Project” on page 33).

Linking with a Library that is Built with Your Project

You can add a library and its source files so that the library gets rebuilt as needed every time you build your project. If the library is the child of a program or subproject, the library inherits settings from the parent program or subproject. However, the library’s source files are built into the library, not into the parent program.

To create such a library:

1. Select the location within the hierarchy where you want to add the library.
2. Click **Add Item into selected** ().
3. Select **Library** from the list of available items and click **Next**.

If **Library** is not in the list of available item types, libraries are not supported in the location that you have selected.

4. Set the name and location of the library project file.

It is recommended that you put your library's project file in the same directory as its source files, and that the name of the project file is the same as the library's name (but with a **.gpj** extension).

5. Click **Finish**.
6. Add source files (see "Adding New Items to Your Project" on page 29).



Warning When including the same **Library** project in multiple programs, be sure that they do not inherit different options. For example, if one program is built with optimizations, and the other is not, your library will contain inconsistently built objects depending on which program was being built. One way to avoid this is to only set options in your **Top Project** and explicitly override any inherited options in the **Library** project.

Linking with a Pre-Built Library

To link a pre-built library into a program, add the library file to the program project file.

1. Select the project file of the appropriate program.
2. Select the appropriate **Add File** action.
3. In the file chooser, browse to the library file and click **Add**.



Note Use this method when the source code for the pre-built library is unavailable. If you do have the source code for the library, use a **Library** project instead, because MULTI will rebuild the source files as necessary.



Warning Never manually add Green Hills-provided target libraries to your project (for example, `libind.a`). The compiler will link in the correct target libraries for your configuration when the appropriate Builder options are set.

If the program pulls in the library through `-llibrary_name`, **gbuild** tries to find a library project that produces the referenced library. It uses the paths specified with `-L library_dir` options to construct the library's full path. If a match is found somewhere in the build tree, **gbuild** builds the library before linking the original program, if necessary (see “Implicit Dependency Analysis” on page 505).

Configuring Projects That Are Compatible Across Multiple Hosts

If you plan to open your project on hosts that run different operating systems, always use forward slashes (/) when specifying paths or filenames. If you always use forward slashes, MULTI tools interpret paths correctly on all supported operating systems. If you configure MULTI to use any third-party tools or send commands to your operating system, follow the path conventions for those tools when specifying the commands or options that use them.



Note The **Project Wizard** creates projects that use your host operating system's path delimiter. If you created your project in Windows using the **Project Wizard** and want it to be compatible across multiple hosts, you may need to open your project files and replace backslashes in paths with forward slashes.

Green Hills Project Files

Project files are text files that list program source files, subprojects, and other files that make up an executable or a project. Each source file is listed along with its type in brackets ([]) if that type cannot be discerned by the Project Manager from the file's extension. Source files and other project file entries can be followed by the project options used when building that file. Project options appear indented on subsequent lines in the project file.

Various special options can only be set in the Top Project file. These include, but are not limited to, options specifying the target architecture and operating system you are building for, customization files you are using, and macro

definitions. For information about setting these options via the GUI, see “The Target Selector Dialog Box” on page 71 and “Setting Build Macros” on page 60.

Except for the text on the first line of the file, which must exactly match `#!gbuild`, text between a `#` character and the end of that line is treated as a comment and ignored. The `#` character may occur in the middle of the line.

For information about advanced Green Hills Project File features, see Appendix B.



Note The Green Hills Project (**.gpj**) file format used by the Project Manager differs from the format of build files (**.bld**) used by previous tools releases.

File Types

You may encounter the following types in the Project Manager **Project Tree**, or in the type definition in a project file:

- Project Files (**.gpj**):
 - **Auto Include** — Pulls in contents of a specific directory to a specified project. This file type cannot specify any children, and must contain the **:autoInclude** option. While this file behaves similarly to a project file, it has the **.auto** extension. For more information about **Auto Include**, see “Automatically Including Files” on page 41.
 - **Library** — A project file to output a compiled library.
 - **Program** — A project file to output an executable.
 - **Project** — General project file. The **Project Wizard** creates the project file **default.gpj** as the top file in the project hierarchy. These general purpose containers can also be used to group **Programs** and **Libraries**.
 - **Reference** — A file type that allows you to import a project subtree anywhere in the build hierarchy without forcing it to inherit the options of the project that it is included in. A **Reference** project file must begin with the **[Reference]** tag, include a single **:reference** option (see “Project Options” on page 273), and must not have any children. When you build your project, the subtree is considered a child of the **Reference** project, but it does not inherit the **Reference** project’s options or those of its parents.

- **Relocatable Object Entry** — A project file to output a relocatable object containing the compiled code from one or more source files.
- **Select One** — A project file that contains multiple source files, only one that is used in the build. These projects are used for multiple-architecture hierarchies (see “Building Platform-Specific Programs from the Same Source Files” on page 48).
- **Shared Object** — A project file to output a shared library. These projects are only available for some target operating systems.
- **Singleton Library** — A project file to output a library containing a single object file. The library is forced into the final executable.
- **Subproject** — A project file to group files within a project, but which does not output anything itself. **Subprojects** differ from **Projects** in that files contained within them are included in the link of their parent project. **Subprojects** are displayed as **Projects** in the Project Manager. This is the default file type for files with a **.gpj** extension.
- **Top Project** — The top file in the project hierarchy. The file type is “Project”.
- **INTEGRITY and velOSity Project Files (.gpj):**
 - **Dynamic Download** — A project file that contains one or more virtual AddressSpaces which can be downloaded via MULTI onto an INTEGRITY kernel already running on your target.
 - **Kernel** — A project file to output a stand-alone executable linked with INTEGRITY libraries suitable for running directly on a target.
 - **Monolith** — A project file that contains a kernel and virtual AddressSpaces and outputs a stand-alone executable, suitable for running directly on a target.
 - **Virtual AddressSpace** — A project file to output an executable which will run in a memory-protected environment on an INTEGRITY system.

For more detailed information about these application types and other INTEGRITY specific types, see the *INTEGRITY Development Guide*.

- **Source and Object Files:**
 - **Assembly (.s, .asm, .ppc)** — Assembly language source files.
 - **C (.c)** — C source files.
 - **C++ (.cc, .cpp, .cxx, .c++, .C, .CXX, .CPP)** — C++ source files.

- **Header (.h, .hh, .H, .h++, .hxx, .hpp)** — C and C++ header files.
- **Linker Raw File** — A file containing raw data that the linker imports into the final program.
- **Object (.o)** — Object files.
- **Prebuilt Library (.a)** — Library file of object modules. These are usually used to specify binary libraries when there is no source available.
- Target Resource Files:
 - **Board Setup (.mbs, .dbs)** — Setup script file for initializing your target board.
 - **Linker Directives (.ld, .lnk)** — Linker directives file that contains a memory map for the target board and a section map for the program. For more information, see “Working with Linker Directives Files” on page 63.
 - **Target Connections (.con)** — Target configuration information for connecting to a simulator or to your target board through a debug server. Double-click this file to open the **Connection Organizer** (see Chapter 4, “Connecting to Your Target” in the *MULTI: Debugging* book).
 - **Script (.rc)** — MULTI command scripts.
- INTEGRITY files:
 - **.bsp** — BSP Description.
 - **.idl** — Object Interface.
 - **.int** — Integrate file.
- Documentation:
 - **Portable Document Format (.pdf)** — Adobe PDF files.
 - **Text (.txt)** — Generic text files. This type is used by default for unclassified file extensions.

Automatically Including Files

The Project Manager supports an **Auto Include** file type that allows you to include a variable number of files matching specified name patterns. When the Project Manager loads an **Auto Include** file, it searches the current directory for files matching any of the file patterns specified by the **:autoInclude** option. The files become the children of the **Auto Include** file for the current build.

Subdirectories are not searched; thus, file patterns that include a relative or absolute path to a set of files in another directory are ignored.

You can create a project file that allows you to automatically pull in files from a particular directory rather than listing specific items to include. To create such a project:

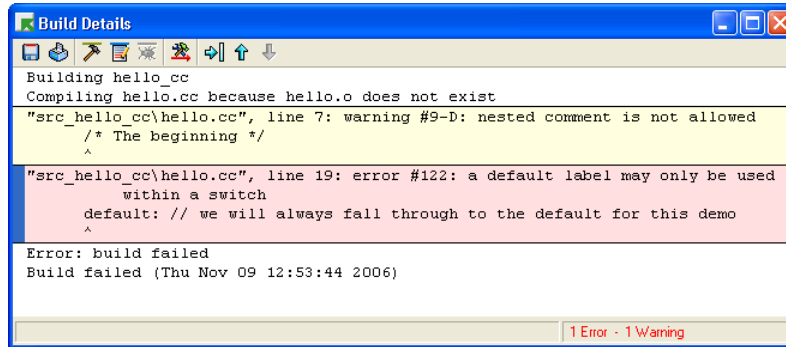
1. Select **Edit** → **Advanced** → **Create Auto-Include Subproject**.
2. In the **Name of new Auto Include** box, enter a location to create a new project file.
3. In the **File patterns to include** box, type a name pattern that matches the files you want to include. The name pattern must not contain any spaces. For example:
 - To match all **.c** files, type ***.c**.
 - To match all files with the name **plugin_n.gpj**, type **plugin_?.gpj**.
 - To match both of the above patterns, type ***.c,plugin_?.gpj**, without any spaces.

Building Your Project

You can instruct the Project Manager to build your entire project, a specific subproject, executable, or library, or an individual source file:

- To build your entire project, select the Top Project (usually named **default.gpj**) in the **Project Tree** and click the **Build** button (.
- To build a specific subproject, executable, or library, select the appropriate project file and click the **Build** button or right-click and select **Build selected file**.
- To build an individual source file, right-click the file and select **Compile selected file**.
- To build multiple files, select the files you want to build, then right-click and select **Build Selected Files**.

In each case, the Project Manager invokes all of the tools necessary to build your project. You can follow the progress of the build in the **Status Tab** at the bottom of the Project Manager window. If the build is successful, its completion is reported here. If errors occur, a **Build Details** window opens:



You can modify or obtain extra information about the next build process by selecting **Build** → **Advanced Build** to open the **Advanced Build** dialog box (for full documentation, see “The Advanced Build Dialog Box” on page 74):

- To remove any previous output files before beginning the build, select the **Clean all output before building** check box, and click the **Build** button.
- To see a dry-run of the build, without actually executing any commands or modifying any files, select the **Show build without execution** check box and click the **Build** button.
- To see the commands that the Builder executes as it builds, select the **Show tool commands** check box and click the **Build** button.

Using the File Shortcut Bar

The **File Shortcut Bar** is located directly above the project tree in the main Project Manager window.



Use the **File Shortcut Bar** to find or build files that match a search string. Your search string can contain wildcards (for example, * and ?). The bar has two components; a drop-down list that enables you to select which action the Project Manager will perform, and a text box, where you specify your search and any options. The Project Manager always searches your entire Top Project for files that match the search string. The following table lists each action, along with the appropriate syntax for the text box:

Build: options files

Builds any *files* specified. You can also specify **gbuild options** (see “The gbuild Utility Program” on page 499).

For example, to delete all previous output files and build the projects **foo.gpj** and **bar.gpj**, select the **Build:** setting and enter:

```
-cleanfirst foo.gpj bar.gpj
```

Find: file

Locates *an instance* of the specified *file* in the project tree. This option is best suited to finding **.gpj** files or uniquely named files quickly.

Next: file

Locates *the next instance* of the specified *file* in the project tree. This option is best suited to finding files with commonly used names.



Note When you specify a *file*, do not include a path. The **File Shortcut Bar** will search for:

- Project files (for example, **foo.gpj**)
- The executable output of a project file (for example, **foo**)
- C (or other source) language input files (for example, **foo.c**)
- The object file output of a source language file (for example, **foo.o**)

You can use wildcards (for example, * and ?) in your **File Shortcut Bar** search string. For example, selecting **Find** and entering ***svc*** locates all projects and source files whose names contain **svc**. If there is exactly one matching file, the Project Manager selects that file in the project tree. If there is more than one match, the Project Manager opens a **Find Results** window with a list that contains each matching file’s name, type, parent project, and path.

If there are no matches for your search string, the Project Manager will append a * wildcard to the end of the string and search again. For example, if you try to find **svc** and there are no files named **svc**, the Project Manager performs your search again as **svc***. It will then select the one file in your project beginning with **svc**, or display the **Find Results** window with a list of files that begin with **svc**.

You can also use wildcards when the **Build** action is selected in the shortcut bar. The behavior is the same as the preceding, except that instead of opening a **Find Results** window, the matching file or files are built. If there are no

matches for your search string, the Project Manager will append * to the string and build all matching files.

Navigating the Project Tab

The **Project** tab of the advanced views pane displays the important components in your project using a layout that is similar to what you might draw on a white board. Projects (**.gpj**) have a small plus or minus sign that allows you to expand or collapse their contents. The **Project** tab displays the current selection in the **Project Tree**. If you select multiple items, the display does not change.

Double-click any item in the **Project** tab to configure that item. For example, if you double-click a project, MULTI opens the project's settings. If you double-click a source file or text file, MULTI displays a dialog that allows you to rename the file.

Navigating the Memory Layout Tab

The **Memory Layout** tab of the advanced views pane is a graphical representation of section maps defined by your executable.

You can navigate the **Memory Layout** pane by using the following buttons:

-  — Zooms in.
-  — Zooms out.
-  — Displays the detailed memory layout data about each section. The data is equivalent to the information output by the **gdump -map** command.
-  — Displays all of the program sections for the selected project file.

You can navigate the colored chart area by clicking and dragging on the colored program sections in the active window. By performing this action, you are zooming in on a specific range of memory. You can also drag the currently displayed range on the scale to view different parts of memory.

Click  to display the **Memory Layout Data** dialog box, which lists all program sections that can be viewed. In the **Memory Layout Data** window, you can single click a section to scroll to that section, or you can double-click a section to zoom into that section. For information about memory layout and

program sections, see “Configuring the Linker with Linker Directives Files” on page 437.

Searching for Project Files

If your project has many files, it may be difficult to locate a particular file within the hierarchy. To search for a file:

- Use the **File Shortcut Bar** by selecting either **Find:** or **Next:**, enter the name of an input (for example, **hello.c**) or output (for example, **hello.o**) file in your project and press **Enter**.



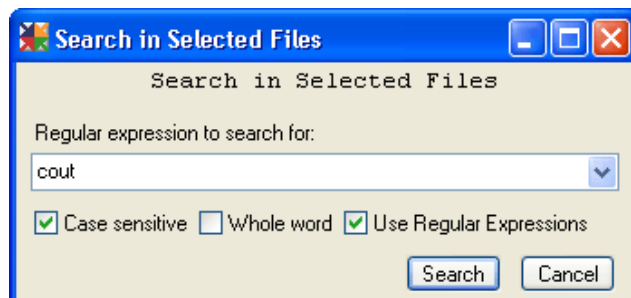
For more information, see “Using the File Shortcut Bar” on page 43.

- You can also search the visually expanded portion of the project tree for options and files. Press **Ctrl+F** and type a search string to search forwards, or **Ctrl+B** and type a search string to search backwards. The Project Manager searches incrementally as you type. Press **Ctrl+F** or **Ctrl+B** to look at the next or previous instance of a string. To exit the search mode, press **Esc**.

Searching Files in the Project Manager

The Project Manager supports searching for a text string in all the searchable files in a project. If you select an individual source file, only that file is searched. If you select a **.gpj** file, all the files referenced by that file are searched.

To begin a search, select **View → Search in ...** to open the **Search in Selected Files** dialog box.



Enter the search string in the text box (or use the drop-down list to select recent search strings). You can also select any of the following searching options:

- **Case sensitive** — The search only finds text that matches the case of the search string exactly. If this box is not selected, the search ignores case when searching for a match. This box is selected by default.
- **Whole word** — The search only finds text that contains the search terms as words. This means that the matching string must be preceded by a non-word character and followed by a non-word character, where word characters are letters, digits, and the underscore. For example, if you select this check box, a search for `ice` does not match `slice` or `ice__`, but it does match `ice-9`. This box is cleared by default.
- **Use Regular Expressions** — The search treats the text you enter in the text field as a regular expression. If this box is cleared, the search treats the text you enter as a fixed string. This box is selected by default.

After you have entered the search string and set your searching options, click **Search** to perform the search. A **Search in Files Results** window opens when the search is complete. For a description of this window, see “Viewing Search in Files Results” in Chapter 2, “The MULTI Editor” in the *MULTI: Editing Files and Configuring the IDE* book.



Note The Search in Files feature uses the BSD **grep** utility. A copy of BSD **grep** is installed along with MULTI. However, BSD **grep** is not part of MULTI and is not distributed under the same license as MULTI. For more information about the license under which BSD **grep** is distributed, refer to the file **bsdgrep.txt**, which is located in the **copyright** subdirectory of the MULTI installation. For information about the search expression format that BSD **grep** uses, refer to the OpenBSD `re_format(7)` man page.

Choosing a Build Mode

MULTI uses one of two different modes for building your projects: parallel or single-thread. To choose a build mode, from the toolbar, select **Tools** → **Options**. In the **Options** window that opens, select the **Project Manager** tab, and locate the **Build Options** section. Select one of the following radio buttons:

- **Parallel Build** — Splits the work of building your program among several processes on your local machine, which speeds up the build as a whole by interleaving independent compilations.

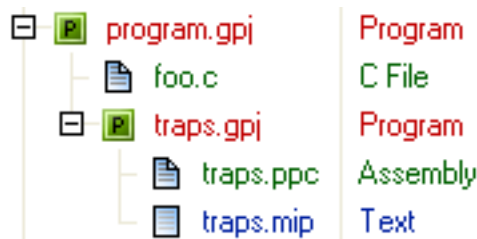
- **Single-Thread Build** — Builds your program using a single process on your local machine. If you have a slow machine that does not perform well in parallel build mode, use single-thread mode. This is the default build mode.

For more information about the Project Manager options tab, see “Project Manager Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

Building Platform-Specific Programs from the Same Source Files

When a program works on multiple hardware platforms, the source files are often identical except for one or more assembly language routines that vary from processor to processor. In cases like this, you can create **Select One** projects that contain the processor-specific files so you can build the same program for multiple target processors.

Consider the following example:



During the build, the **Select One** project **traps.gpj** uses only one processor-specific file, **traps.ppc** or **traps.mip**, depending on the specified target processor.

If you have multiple files that are specific to a single processor, create multiple **Select One** projects. Suppose the program in this example also uses **bdriver.ppc** and **bdriver.mip**. In this case, you would create another **Select One** project, **bdriver.gpj**.

By default, the Project Manager chooses the file in the **Select One** project with an extension appropriate to your target architecture. If it cannot find a target-specific file, it uses the file with the **.c** extension. To override this behavior, set the **Advanced**→**Project Options**→**'Select One' Project Extension List** option. This option allows you to specify a list of extensions in order of priority. Usually, you set this option in your Top Project, and it is

only necessary in advanced project configurations. For more information, see “**Select One’ Project Extension List**” on page 284.

Creating Your Project For Multiple Platforms

1. Create the project for your program.
2. Add all of the source files except the processor-specific assembly files to the project.
3. Add a **Select One** subproject (for example, **traps.gpj**) to the project. For more information, see “Adding New Items to Your Project” on page 29.
4. In the **Project Tree**, add the processor-specific assembly files to the **Select One** project.
5. If necessary, set the **’Select One’ Project Extension List** option in your **Top Project**.

Setting Builder Options

The Project Manager provides many options to control such features as:

- Extensions or restrictions to C or C++
- The location of your libraries and header files
- The level of debugging information to be generated
- How your project will be optimized
- Run-time error checking
- Program data allocation (and whether any Special Data Area is utilized)

Each of the Builder options corresponds directly to one or more of the command line compiler driver options. There are four different types of Builder options:

- **Switches** — Two or more settings from which you choose only one (for example, **Optimization Strategy** and **Debugging Level**).
- **Combination Switches** — Multiple settings from which you choose all, none, or any combination (for example, **Run-Time Error Checking** and **MISRA C**).
- **Strings** — Accepts a single string value (for example, **Output File Name** and **Start Address Symbol**).

- **Lists** — Accepts multiple string values (for example, **Include Directories** and **Library Directories**).
- Builder options are set through the **Build Options** window. For more information, see:
 - “The Build Options Window” on page 50 — for an overview of the **Build Options** window.
 - Chapter 5, “Developing for PowerPC” — for information about the main PowerPC features that can be accessed or controlled with options.
 - Chapter 6, “Builder and Driver Options” — for a complete list of available options.

The Build Options Window

To open the **Build Options** window, right-click the file that you want to set the options in, and select **Set Build Options**. This window has three tabs, which allow you different ways of viewing Builder options, and opens on the last tab you viewed. A fourth tab, **Deactivated**, only appears if your project contains any build options that are marked as disabled for your architecture (see “CommandOption Definitions” on page 847). Note that the **Deactivated** tab does not appear if the disabled option is marked with the `{optional}` conditional control statement (see “Conditional Control Statements” on page 837).

To set or edit an option on any of the tabs, double-click it and select a setting, or enter a value in the dialog.

The present value or setting of the option is displayed in the **Value** column. Options can be set at different levels of the Build hierarchy. An option set in a **.gpj** file is inherited by all the files contained in that **.gpj** file (see “Inheriting Options from Parent Build Files” on page 62). Each value appears in one of three colors:

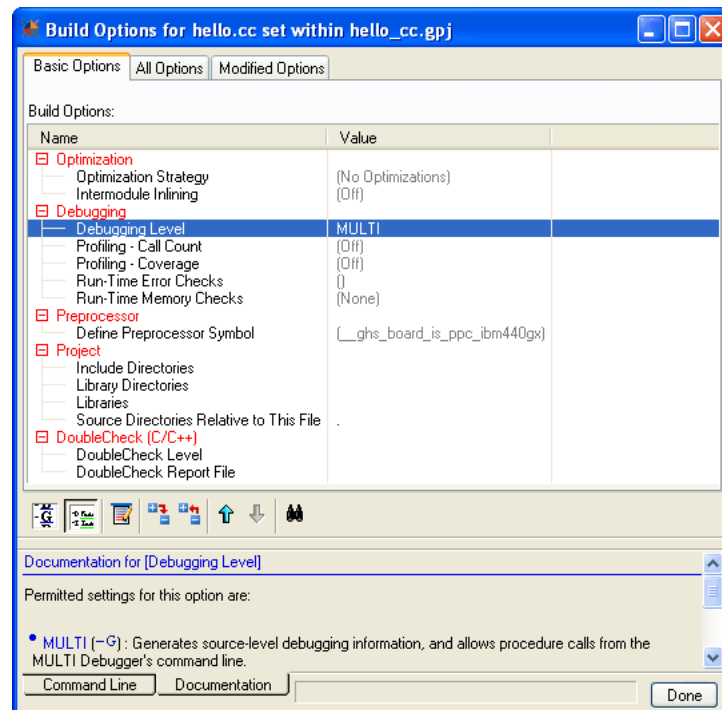
- Values printed in gray and that appear in parentheses are default values that have not been explicitly set anywhere in the project. These values are listed as **Not Set** when you right-click or double-click the option. Note that not all options have default values, and that some defaults can change depending upon other option settings. For example, setting **Optimization**→**Optimization Strategy** for a project may affect the default

setting for **Optimization**→**Optimization Scope**→**Unroll Larger Loops** (and many other options) on any of that project's files.

If you open the **Build Options** window for a file that does not produce build output, such as your Top Project, the **Value** column does not display default values.

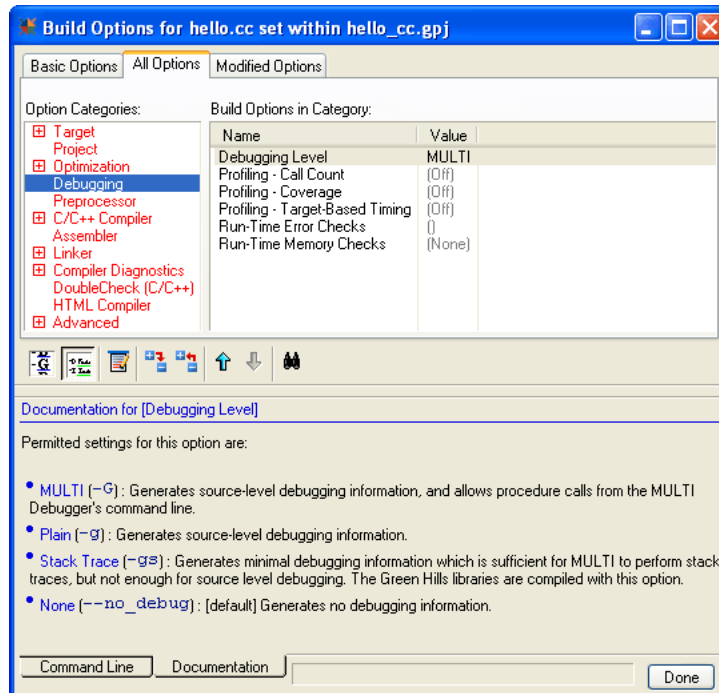
- Values printed in black are inherited from a parent file. These values are listed as **Inherited** when you right-click or double-click the option. You can override inherited values by setting the option to a new value in the present file.
- Values printed in blue are set in the present file.

The Basic Options Tab



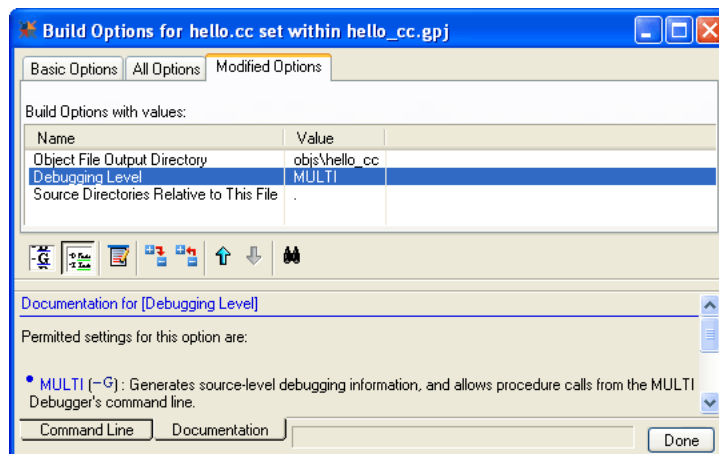
This tab lists only the common options that most users will need to control. Categories can be expanded or contracted by clicking the plus or minus sign to their left. Double-click an option to edit it.

The All Options Tab



This tab lists all of the Builder Options, which are categorized by type or associated tool. The pane on the left displays the category structure. Click a category to display the options it contains in the pane on the right.

The Modified Options Tab

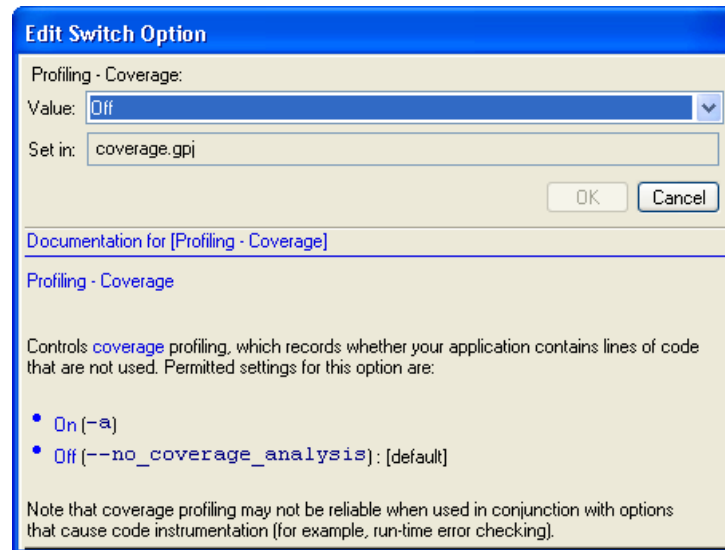


This tab displays those options that have been set in, or are inherited by, this file, together with their current settings. If you right-click the option that you

want to modify and select **Goto Option Text**, the MULTI Editor opens on the line in the project file where that option is set. For options that are set in multiple locations in the project tree, the opened file is the closest parent project where the option is set.

Setting a Switch Option

When you double-click a switch option, a dialog box like this appears.



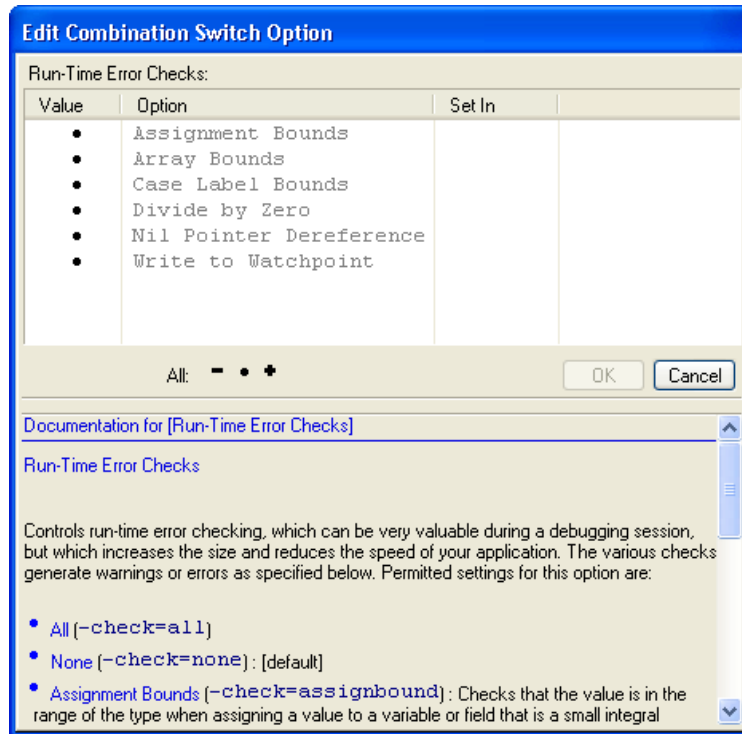
If the option has already been set at this or a higher level of the project hierarchy, its present setting is displayed in the **Value** field. The file where the option was set is displayed in the **Set In** field.

To set (or override a previous setting for) the option:

1. Select a value from the drop-down list. The **Set In** field changes to display the current file's name.
2. Click **OK** to save the new setting and return to the **Build Options** window.

Setting a Combination Switch Option

When you double-click a combination switch option, a dialog box like this appears, where all of the available sub-options are displayed:



If a sub-option has already been set at this or a higher level of the project hierarchy, its present setting is displayed in the left-hand column. The file where the sub-option was set is displayed in the **Set In** column. Sub-options set higher in the project hierarchy are printed in black and have an arrow pointing up and to the left in the left-hand column. Those set in the present file, are printed in blue, and those that have not been set are printed in gray.

To set (or override a previous setting for) a sub-option:

1. Click the option that you would like to change. These symbols toggle between:
 - ■ — not set
 - + — enabled
 - ■ — disabled

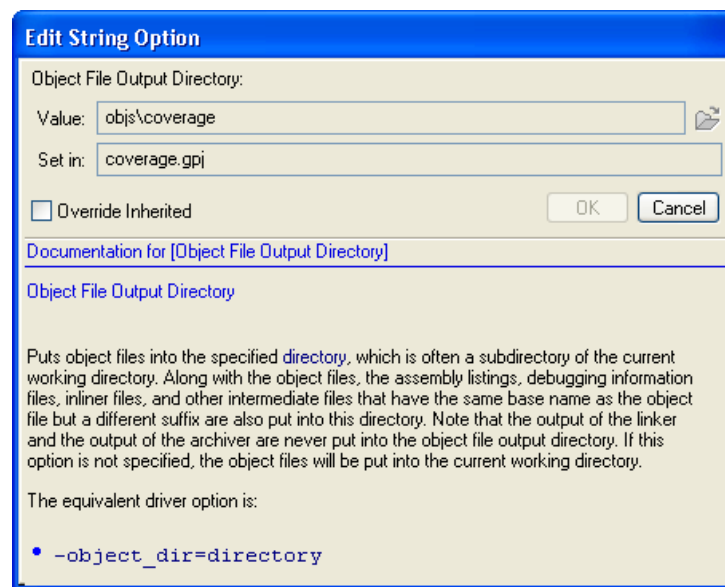
The line for the sub-option turns blue, and the **Set In** field changes to display the current file's name.

You can enable or disable all of the sub-options with the **All** toggle (**All:**   ) option.

2. Edit other sub-options as appropriate and click **OK** to save the new setting and return to the **Build Options** window.

Setting a String Option

When you double-click a string option, a dialog box appears:



If the option has already been set at this or a higher level of the project hierarchy, its present setting is displayed in the **Value** field. The file where the value was set is displayed in the **Set In** field, and the **Override Inherited** check box is displayed.

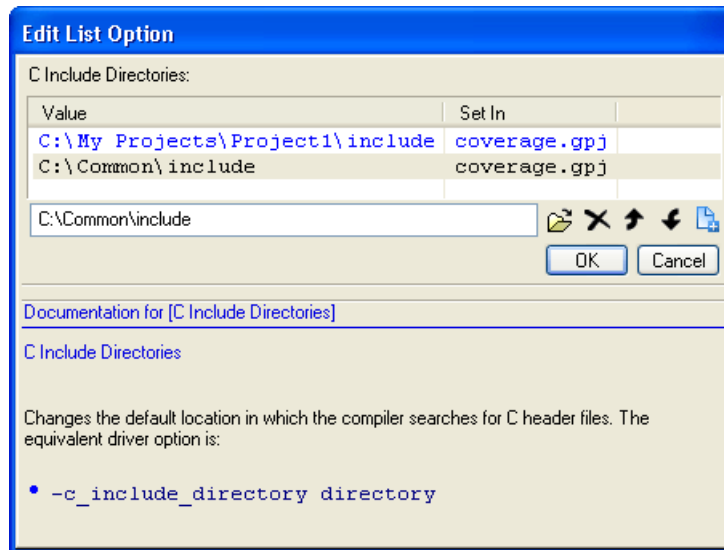
To set the option:

1. Enter an appropriate string in the **Value** field (if the option has already been set, you must select the **Override Inherited** check box before you can access the **Value** field). If the option requires a directory or filename as a value, a folder icon () is displayed, which you can use to navigate to the appropriate file or directory.

2. The **Set In** field changes to display the current file's name. Click **OK** to save the new setting and return to the **Build Options** window.

Setting a List Option

When you double-click a list option, a dialog box appears:



If the option has already had values assigned to it at this or a higher level of the project hierarchy, its present settings are displayed in the **Value** field. The file where each value was set is displayed in the **Set In** field.

To delete a value from the list:

- Click the value you want to delete, and click the delete button (✕).

To add a value to the list:

1. Click the **Add** button (📁) and enter an appropriate string in the text field. If the option requires a directory or filename as a value, a folder button (📁) is displayed, which you can use to navigate to the appropriate file or directory.
2. The new value is appended to the bottom of the list and the current file's name appears in its **Set In** field. Click **OK** to save the new setting and return to the **Build Options** window.

To reorder the list:

1. Click the value you want to move to select it.
2. Click the **Move Up** button () to move the value up the list, or the **Move Down** button () to move the value down the list.
3. Click **OK** to save the new order and return to the **Build Options** window.

The Build Options Window Toolbar

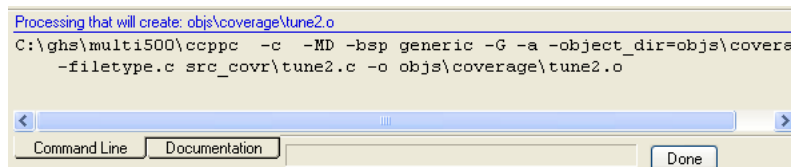
The following buttons are available on the **Build Options** window:

-  (**Show Driver Options**) — Toggles the display of driver option names.
-  (**Show Description**) — Toggles the display of option descriptions.
-  (**Edit Selected Option**) — Opens the project in the MULTI Editor where the selected option is set.
-  (**Expand All**) — Displays all sub-categories.
-  (**Contract All**) — Hides all sub-categories.
-  (**Goto Parent**) — Goes to the options for the parent of the current file.
-  (**Goto Child**) — Goes to the options for the child of the current file.
-  (**Search for Options**) — Opens the **Search** dialog box, which allows you to perform text searches on the option category tree.
-  (**Close**) — Closes the **Build Options** window. Whether this button appears on the toolbar depends on the setting of the option **Display close (x) buttons**. To access this option, select **Tools** → **Options** → **General Tab**.

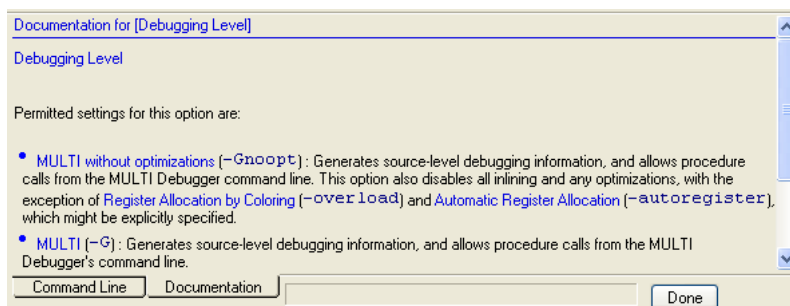
The Build Options Window Lower Tabs

Two tabs are available from the lower portion of the **Build Options** window:

- The **Command Line** tab, which shows the driver options that will be passed to the compiler when you build the selected file:

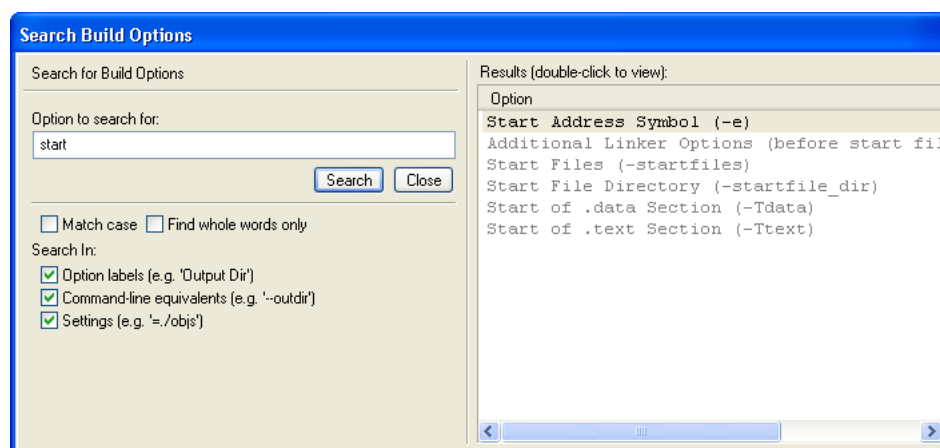


- The **Documentation** tab, which provides documentation for whichever build option is currently selected:



Searching For Options

If you are unable to find the option you want to set, click the search button (🔍) to open the **Search Build Options** dialog box:



Enter an option name or part of a name in the **Option to search for** text field, and click the **Search** button. By default, the Project Manager searches all three of:

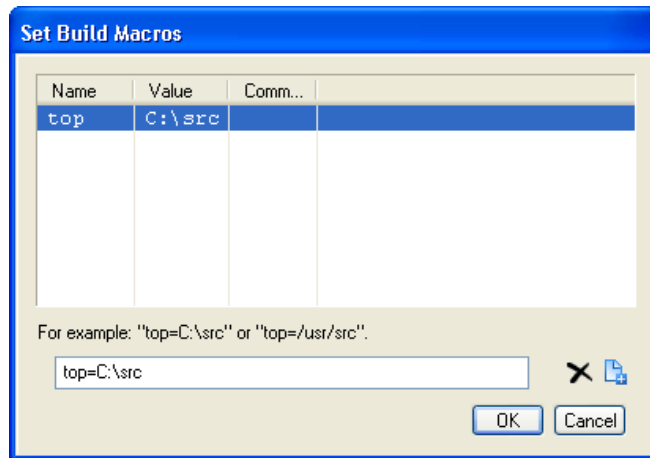
- **Build Option Names**
- **Compiler Driver Option Names**
- **Settings**

Search results appear in the right-hand **Results** pane. Double-click a result to go to its entry in the category tree and close the search dialog.

Setting Build Macros

You can define macros to stand in for commonly used strings in option settings.

To set a macro, from the main Project Manager window select **Edit** → **Set Build Macros** to open the **Set Build Macros** dialog box:



To define a macro, enter *macro_name=value* in the text field. *value* can be the value of a builder option, or part of the value of a builder option.

After you define the macro, use *\$macro_name* anywhere in a builder option where you want the Project Manager to substitute the value of *macro_name*.



Note The predefined macro `GHS_TOOLS_DIR` specifies the location of your MULTI installation.

Example 1. Defining and Using a Macro

This example explains how to define a macro for the root of the project on Windows hosts. The root directory of the project in this example is `C:\src`. To define a macro called `top` to represent the root directory:

1. Open the **Set Build Macros** dialog box.
2. Enter: `top=C:\src` and click **OK**.

The include files for the project are located in `C:\src\include`. To specify this include directory to the Project Manager using the `top` macro:

1. Click **Edit** → **Set Build Options** to open the **Build Options Window**.

2. Navigate to the **Project→Include Directories** option and open it.
3. Enter `$top\include` and click **OK**.



Note You can also define a macro to be the name and value of a driver option. To use a macro defined in this way, you must edit the project (**.gpj**) file in a text editor and add `$macro_name` at the location where you would normally specify the option.

Importing Environment Variables

You can import variables from your environment to stand in for strings used in your option settings.

To import a variable:

1. Click **Edit → Advanced → Set Imported Environment Variables** to open the **Import Environment Variables** dialog box.
2. Enter the name of the environment variable you want to import. The current value from the environment is displayed in the **Value** column of the dialog box.
3. Click **OK** to import the variable.



Note You can only import an environment variable if it is defined in your environment. If you previously imported an environment variable in a project and you open that project when the variable is not defined, MULTI issues a warning.

After you define the environment variable, use `$variable_name` anywhere in a builder option where you want the Project Manager to substitute the value of `variable_name`.

If the environment variable stores the name and value of a driver option, you can edit the project (**.gpj**) file in a text editor and add `$variable_name` at the location where you would normally specify the option.

The Project Manager tracks changes to the environment variables you import so that anything in your project that uses an environment variable is rebuilt whenever you modify that variable's value.

Example 2. How to Set Imported Environment Variables

To include header files in the **headers** subdirectory of a third-party SDK installed in a directory specified by the environment variable `SDK_DIR`:

1. Open the **Set Imported Environment Variables** dialog box.
2. Enter `SDK_DIR` and click **OK**.
3. Click **Edit** → **Set Build Options** to open the **Build Options** window.
4. Navigate to the **Project** → **Include Directories** option and open it.
5. Enter `$SDK_DIR\headers` and click **OK**.



Note Macros and imports are only valid if they are set in the Top Project. If you use the GUI for setting builder option macros or importing environment variables, the macros and imports are always saved in the Top Project. This behavior is different from the **Set Build Options** dialog box where the options are set on whichever file is selected.

Inheriting Options from Parent Build Files

A file inherits its options from its parent project (**.gpj**) file and all parents of that project. You can override these inherited options by setting options in the file itself. When the Builder compiles your project, the Top Project options are applied first, with each subsequent child project adding its options to the existing ones, overriding or appending where specified, until finally the individual file options are added for each file.

Setting Options for Projects and Subprojects

Because you can reuse project files for programs, subprojects, and libraries in multiple projects, it is important to specify whether the options for a project file are specific to the current project, or whether the options are specific to the project file itself regardless of the project to which it belongs.

- If the options apply everywhere that the child project is used, select that project (**child.gpj**), and set the options. These options are written to the child project file, and therefore are set for that project file regardless of what parent it currently belongs to.
- If the options apply only when the child project is the child of a particular parent project (**master.gpj**), select the child project (**child.gpj**) choose

Edit → **Advanced** → **Set Options in Parent**, and set the options. These options are written to **master.gpj** and apply only to **child.gpj**. If you reuse the child's project file in a different project, the options are not carried over into the new project.

Setting Options for Source Files

You can set options for an individual file or for a list of files.

- To set an option for an individual file, select the file and set the options.
- To set an option for a group of files, select the project file that contains the files in the Project Manager's hierarchy of your project, and set the options. The options are set in the project file. All files below the project file, including its source files, will inherit these options.

Working with Linker Directives Files

A linker directives file has a **.ld** extension and controls how the linker links your program and prepares program sections for allocation into specific regions of memory on your target. A typical linker directives file contains:

- *A memory map* — Lists the starting address and size of each region of memory that is used to contain program code.
- *A section map* — Lists the program sections and specifies the memory regions that they will be loaded into.

Linker directives files can also define constants and special symbols that the linker uses when linking and loading your executable. For more information about the file format, see “Configuring the Linker with Linker Directives Files” on page 437.

One or more of the following default linker directives files are generated automatically when you create a new project for the **Stand-alone** operating system and are placed within the Target Resources project:

- **standalone_ram.ld** — The program is linked so that it is loaded into, and executes out of RAM. Such programs are usually downloaded and started by running the Debugger and a debug server. This layout is generally used in the early stages of development.

- **standalone_romrun.ld** — The program is linked so that it can be loaded into, and executes out of ROM/FLASH. Such programs must be burned into flash (for example, by using MULTI Fast Flash Programmer). This layout may be used for producing a final executable. It requires the least amount of RAM, but is typically slower, and cannot be debugged using software breakpoints. For more information about the MULTI Fast Flash Programmer, see Chapter 20, “Programming Flash Memory” in the *MULTI: Debugging* book.
- **standalone_romcopy.ld** — The program is linked so that it can be loaded into ROM/FLASH, but copies itself to RAM and executes out of RAM. Such programs must be burned into flash. This layout can be used for producing a final executable. It is faster than the `romrun` layout and offers software breakpoint debugging, but takes longer to initialize and requires more RAM.

Depending on the **Program Layout** setting you choose in the **Project Wizard** (see “Settings for Stand-Alone Programs” on page 31), one of these files is added to your program project file (for example, **hello.gpj**) for use at link-time. To change the **.ld** file used for linking, reconfigure your program project file and select a new **Program Layout** (see “Configuring Existing Items” on page 33).



Note These default files are automatically generated by the **Project Wizard** by combining the generic section maps in the `install_dir/target/ppc` directory with a board-specific memory map taken from `install_dir/target/ppc/target_dir`.

You may need to edit these default files to add new memory regions or sections, or to change their attributes. To edit a file, double-click it (or right-click it and select **Edit**) to open it in a text editor.

The Project Manager GUI Reference

Chapter 3

This Chapter Contains:

- The File Menu
- The Edit Menu
- The View Menu
- The Build Menu
- The Connect Menu
- The Debug Menu
- The Tools Menu
- The Windows Menu
- The Help Menu

This chapter provides a comprehensive description of the commands and options in the main Project Manager window menu bar and describes hot-keys and shortcuts, when available.

The File Menu

The following table lists the selections available from the Project Manager **File** menu:

New Top Project (Ctrl + N) 	Opens the Project Wizard , which guides you through the process of creating a new project. For more information, see “Creating A New Project” on page 22.
Open Project (Ctrl + O) 	Opens a project.
Open Reference BSP Project	Opens the BSP Selector dialog box that allows you to select an installed BSP from your Green Hills operating system installation directory. This BSP project file is opened in a new window as a reference for you to find examples and other files included in the BSP project. For more information about the contents of the BSP project, see the documentation for your Green Hills operating system.
Close Project	Closes the current project.
Save Project (Ctrl + S) 	Saves the current project.
Reload Saved Project	Reloads the current project from disk, discarding any changes made since the last save.
Print Current View (Ctrl + P)	Prints the currently displayed project hierarchy. The hierarchy will be printed in the exact state that it is displayed in the source pane. If you want to print the contents of subprojects, you need to expand them first so that they are displayed.
Print Expanded Project	Prints the fully expanded project hierarchy. Does not change the displayed project hierarchy.

Write Expanded Project to File
Writes the fully expanded project hierarchy to a text file.
Recent Files
This submenu contains recently opened files. Select one to open it in the Editor.
Recent Projects
This submenu contains recently opened projects. Select one to open it in the Project Manager.
New Window
Opens a new Project Manager window with no open project.
Close Project Manager (Ctrl + Q)
Closes the current Project Manager window, prompting to save any outstanding changes.
Exit All
Closes all open windows in MULTI.

The Edit Menu

The following table lists the selections available from the Project Manager **Edit** menu:

Configure
Opens a dialog box to configure the selected component.
Set Build Options
Opens the Build Options window for the selected file. For more information, see “Setting Builder Options” on page 49.

Modify Project

Contains actions appropriate to the selected component. These actions can include:

- **Add Item** — Opens a **Select Item to Add** dialog box. You can use this dialog box to add context-appropriate components to your project.
- **Comment Out** — Marks the item so that it is not included when the project is built. The item is still visible in the Project Manager, but is disabled. This is different from adding a comment to a project file with the # character.
- **Resolve Requirements** — For stand-alone projects, this menu item checks the selected component and its children for dependencies on customized libraries in your target resources project. If those libraries are not present, they are added to your project.

For INTEGRITY projects, this menu item checks for many different types of dependencies specific to INTEGRITY, and attempts to resolve those that are unsatisfied.

- **Uncomment** — Removes the comment mark added by **Comment Out** from the item so that it is included when the project is built. This menu item is different from removing the # character at the beginning of an existing comment.

In INTEGRITY projects, additional options may be available. For more information, see the *INTEGRITY Development Guide*.

Edit (Ctrl + E)

Opens the selected file in an Editor window.

Graphically Edit

This item is only available for some file types.

Opens a customized graphical editor for the file, such as the **Connection Organizer** (see Chapter 4, “Connecting to Your Target” in the *MULTI: Debugging* book).

Add Item into *selected file* (Ctrl + I)

Opens a component selector for you to add a context-appropriate component into the current project. If only one component is appropriate, it will skip straight to the add dialog for that component. The files are added into the selected project.

Add File into *selected file*

Opens a browser window that allows you to choose a file to add into the current project.

To make the project file portable, MULTI uses relative or base names for the added files whenever possible by resolving them relative to the set source directories.

Add File after *selected file*

Similar to **Add File Into *selected file***, except that this menu item is not shown when you have a container project selected, and the file you insert goes immediately after the selected file (in the same parent project).

Undo (Ctrl + Z)

Undoes the last change made to your project by the Project Manager during your current session. You can use this item repeatedly to undo multiple changes.

If you make changes to a file outside of the Project Manager, those changes will be lost if an **Undo** operation affects that file.

Redo(Ctrl + Y)

Re-applies the last change made by **Undo**. You can use this item repeatedly to redo multiple changes made by **Undo**

If you make changes to a file outside of the Project Manager, those changes will be lost if a **Redo** operation affects that file.

Copy selected file as Link

Copies the selected items to the clipboard. Use this item if you intend to **Paste as Link**.

Copy selected file Local

Copies the selected items to the clipboard. Use this item if you intend to **Paste Local**.

Paste as Link

Pastes the clipboard items and creates a reference to them in the selected context.

Paste Local

Pastes the clipboard items and makes a recursive copy of each item within the selected context.

Remove selected file

Removes the currently selected files from the project. The file itself is not deleted.

Delete selected file

Deletes the currently selected files from the project and also from the disk.

Set Build Target

Opens the **Target Selector** dialog box, which allows you to change the board and processor for which your project will be built. For more information, see “The Target Selector Dialog Box” on page 71.

Set Build Macros

Opens the **Set Build Macros** dialog box, which allows you to define macros to stand in for commonly used strings in option settings. For more information, see “Setting Build Macros” on page 60.

Advanced

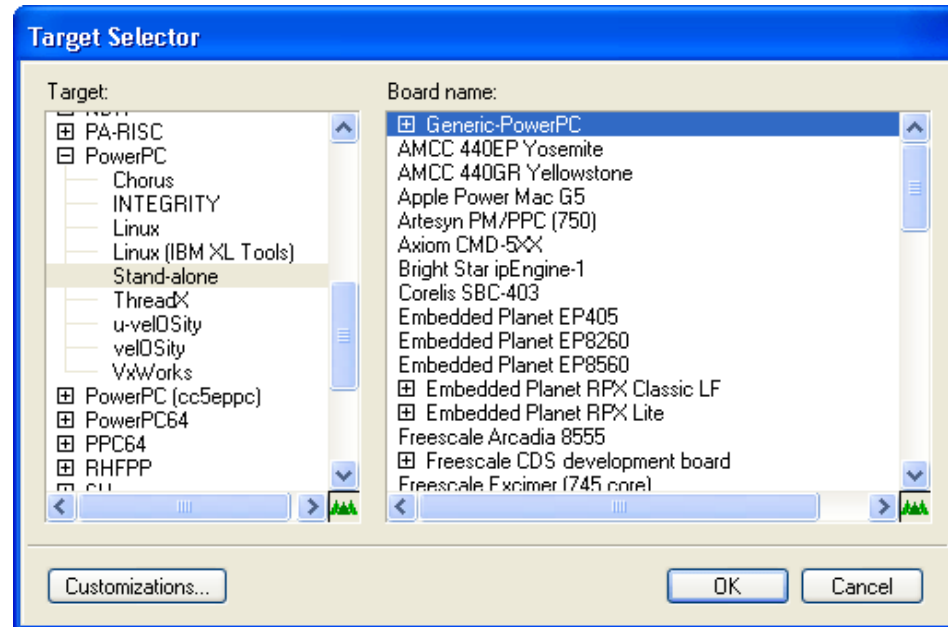
Displays the **Advanced** submenu, which contains the following items:

- **Add File After selected file** — Similar to **Add File Into selected file**, except that this menu item is not shown when you have a container project selected, and the file you insert goes immediately after the selected file (in the same parent project).
- **Create Auto-Include Subproject** — Opens the **Add new auto-include project** dialog box that allows you to specify patterns of files to automatically include in the selected project. For more information, see “Automatically Including Files” on page 41.
- **Set File Type** — Opens the **Set Type** dialog box that allows you to change the file type of the selected file. This operation is not available for project files. For more information, see “File Types” on page 39. For information about compiling files with non-standard file extensions, see “Recognized Input File Types” on page 87.
- **Set Options in Parent** — Opens the **Build Options** window to set Builder options for the selected file, but stored in the parent. For source files, this is the normal behavior provided by **Set Build Options**, but for projects this allows options to be set for a particular occurrence of the project. For more information, see “Setting Options for Projects and Subprojects” on page 62.
- **Set Imported Environment Variables** — Opens a dialog box that allows you to import variables from your environment to stand in for strings used in your option settings. For more information, see “Importing Environment Variables” on page 61.
- **Simplify All Filenames** — Attempts to convert any absolute filenames into filenames relative to the set source directories. This command modifies your project. Specifically, if the path of a file is removed and if the file can still be found by searching the source directories list, then the full pathname is replaced by the filename without a path. This is a simple means of converting absolute pathnames in projects to short relative pathnames, which increases portability. This resolving is done automatically when files are added, so this functionality is usually only useful after changing source directories.

The Target Selector Dialog Box

When you create a project (see “Creating A New Project” on page 22) you must specify a target to build for.

To change your Build Target after you have already created a project, select **Edit** → **Set Build target** to open the **Target Selector** window.



To select a new target:

- Choose a processor family from the **Target** list.
- Choose a board from the **Board Name** list. Some boards are not supported for all operating systems. If your board is not listed, expand the **Generic-PowerPC** item, and select your processor from the list.



Note Advanced users can add customization (**.bod**) files, which may define custom file types or control custom tools. To add such a file, click the **Customizations** button, click the **Add** button in the **Build Customizations** dialog, and select a custom **.bod** file with the file chooser. For more information about customization files, see Appendix C.

The View Menu

The following table lists the selections available from the Project Manager **View** menu:

Search in Source Files in <i>selected file</i> Opens the Search in Selected Files dialog box to perform a full-text search of all source files contained within the selected project file. For more information, see “Searching Files in the Project Manager” on page 46.
Search in <i>selected file</i> Opens the Search in Selected Files dialog box to perform a full-text search of your selected source file. For more information, see “Searching Files in the Project Manager” on page 46.
Search in All Source Files Opens the Search in Selected Files dialog box to perform a full-text search of all source files.
Expand Project Recursively expands the selected file to show all its children.
Contract Project Recursively contracts the selected file to hide all its children.
Show Paths Displays the Show Paths submenu, which offers options to display the following: • Full Paths — Displays the full path information for files in the Project Manager window. • Relative Paths — Displays the relative path information for files in the Project Manager Window. This item is enabled by default. • Filenames Only — Only displays the filenames in the Project Manager Window.
Highlight Selections Toggles the highlighting of all the files that are included within the currently-selected project file. This setting is saved across sessions.
Show All Views Enables or disables the view of the content panes.
Show Type Column Displays the file type in the project tree.
Show Options Column Displays compiler options that are set at this level of the hierarchy.

Show Size Column

Displays the size of executable files and the size of the compiled output of your source files.

Filter Project View

Displays the **Filter** submenu, which offers options to show or hide various file types in the **Project** view pane. Selecting **Show All** makes all files visible.

The Build Menu

The following table lists the selections available from the Project Manager **Build** menu:

Compile *selected file*(Ctrl + F7)

Invokes the compiler on the currently selected source file. Does not link or relink this file into an executable. This is only available if you have a source file selected.

Build *selected file* (F7)

Invokes the compiler, assembler, and linker as appropriate, to build the currently selected files and projects.

Stop Build

Stops the build process. If MULTI is in the process of generating an output file when you stop the build, that output file may be left in an incomplete state. If you have problems building or running your project after using this menu item, clean and rebuild your project.

This item is only available while you are building files.

Preprocess *selected file*

Preprocesses the selected file.

This item is only supported for source files.

Rebuild *selected file*

Builds the currently selected files after first removing the intermediate output files.

Build Ignoring Errors *selected file*

Builds the current project, ignoring any detected errors. Normally, the build stops when an error occurs to prevent downstream failures like a link failing because of missing objects.

Clean *selected file*

Deletes all of the files that are normally created when building the project. This includes object files, libraries, and executables. In other words, at each step where a file would be created in a normal build, the file is deleted instead. The only files that remain will be the source files necessary for building the project from scratch.

Advanced Build

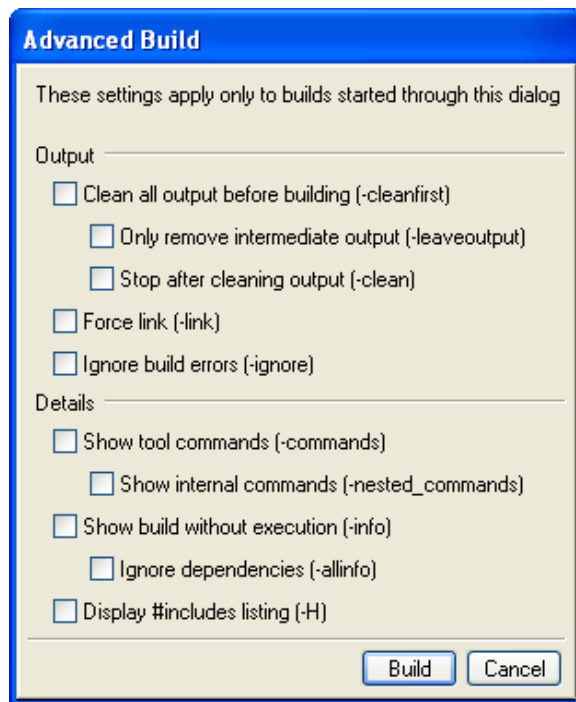
Opens the **Advanced Build** dialog box, through which you can execute builds with less common options. For more information, see “The Advanced Build Dialog Box” on page 74.

View Build Details (F6)

Opens a window displaying all status output from the latest build.

The Advanced Build Dialog Box

To open the **Advanced Build** window, select **Build** → **Advanced Build**:





Note These options apply only to the current build.

Clean all output before building
Deletes all previous output files, and then builds.
Only remove intermediate output
Deletes intermediate output files, and then builds.
Stop after cleaning output
Deletes all previous output files, without building.
Force link
Forces linking. The default behavior is to re-link the executable only if any of its components have changed.
Ignore build errors
Continues building, even if errors are encountered.
Show tool commands
Displays commands as they are executed.
Show internal commands
Displays full commands (including internal commands), as they are executed.
Show build without execution
Simulates the build without executing the commands.
Ignore dependencies
Simulates the build, ignoring dependency information (rebuilding all files even if they are up-to-date), without executing the commands.
Display #includes Listing
Displays a list of files opened by <code>#include</code> directives during the current build only (see “Preprocessor Options” on page 192).

The Connect Menu

The following table lists the selections available from the Project Manager **Connect** menu:

Connect (F4) 
Opens the Connection Chooser dialog box, which allows you to connect to a target or simulator.
Connection Organizer
Opens the Connection Organizer . For more information, see Chapter 4, “Connecting to Your Target” in the <i>MULTI: Debugging</i> book.
Task Manager
Opens the Task Manager , which displays the tasks running on the target. Only available for multitasking debug servers.
Load Module
This submenu is only available for multitasking debug servers. It allows you to download a new object module to the target. Choose Load Module again from the submenu to choose the module to download from a dialog box, or choose one of the recently downloaded modules from the list provided. Load Module is only available when you are connected to a run-mode target that supports and was configured with a dynamic loader (for example, the <code>LoaderTask</code> on INTEGRITY). For more information, see “Establishing Run-Mode Connections” in Chapter 23, “Run-Mode Debugging” in the <i>MULTI: Debugging</i> book.
1 connection
2 connection
3 connection
4 connection
Lists the most recently connected debug servers. To connect to one of them, select it.

The Debug Menu

The following table lists the selections available from the Project Manager **Debug** menu:

Debug <i>selected program</i> (F5) 
Opens the Debugger on the currently selected project.
Debug Other Executable
Opens a file chooser that allows you to select the executable you want to debug.
1 <i>pathname</i> 2 <i>pathname</i> 3 <i>pathname</i> 4 <i>pathname</i>
Lists the most recent programs opened in the Debugger. Select a program to open it in the Debugger.

The Tools Menu

The following table lists the selections available from the Project Manager **Tools** menu:

Configuration Opens a submenu offering the following options: • Save Configuration as User Default — Saves the current configuration into the default user configuration (.cfg) file for MULTI, so that it will be used for future sessions. • Clear User Default Configuration — Deletes the default user configuration file for MULTI. • Save Configuration As — Opens a file chooser dialog box for you to choose a file and then saves the current configuration into it. • Load Configuration — Opens a file chooser dialog box that allows you to load a configuration file. • Set INTEGRITY Distribution — Opens the Default INTEGRITY Distribution dialog box, where you can set the location of your INTEGRITY installation directory. • Set u-velOSity Distribution — Opens the Default u-velOSity Distribution dialog box, where you can set the location of your u-velOSity installation directory.
Editor Opens a file chooser to select a file to open in the Editor.
Launcher Opens the Launcher. For more information, see “The MULTI Launcher” on page 4.
Flash <i>selected program</i> Opens the Write to Flash Memory dialog box, which allows you to write the selected program to flash memory on the target. The Flash menu item is not available if MULTI is connected to multiple targets.
View DoubleCheck Report Opens the default web browser on your system and displays a DoubleCheck report. For more information about this feature, see Chapter 8, “The DoubleCheck Source Analysis Tool”.
Checkout Browser Opens the Checkout Browser.

Use Utilities

Opens the **Utility Program Launcher** dialog box. For more information, see “The Utility Program Launcher Dialog Box” on page 79.

Convert Legacy Project

Opens a file chooser that allows you to select a legacy build (.bld) file to convert into a project (.gpj) file.

Options

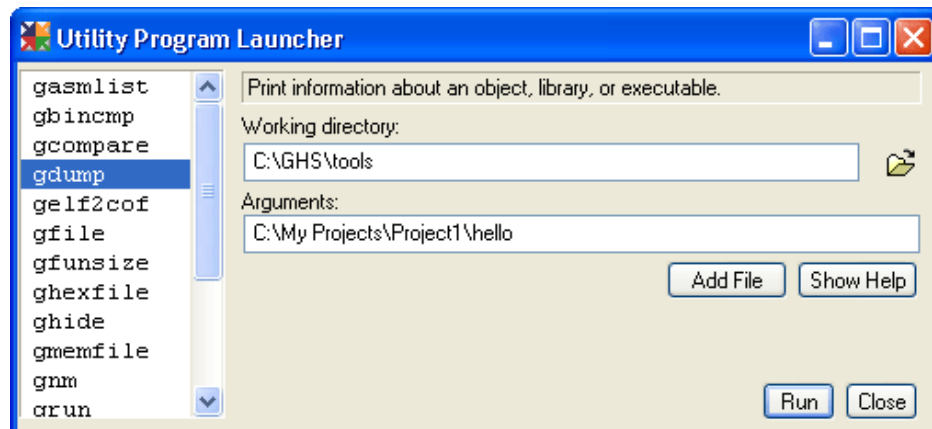
Opens the **Options** dialog box for you to change options that affect the way the Project Manager and other MULTI tools look and behave. For a description of each Project Manager option, see “Project Manager Configuration Options” in Chapter 9, “Configuration Options” in the *MULTI: Editing Files and Configuring the IDE* book.

Customize Menus

Opens the **Customize Menus** window, which allows you to add new menus to the end of the menu bar; add new submenus and menu items to the end of existing menus; and remove menus, submenus, and menu items. For information about how to use this window, see Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

The Utility Program Launcher Dialog Box

To open the **Utility Program Launcher**, select **Tools → Use Utilities**.



To use a utility:

1. Select the utility from the list on the left.
2. Click  to navigate to the **Working Directory** where you will run the command.

3. Enter any **Arguments** you want to pass. As a minimum, you must generally pass the name of the file to be processed. To display information about the syntax of arguments, click **Show Help**.

If you open the **Utility Program Launcher** from the Project Manager while you have an item selected in your project, the path to that item may appear in the **Arguments** box.

4. Click **Run** to run the utility.

For documentation of all the utilities, see Chapter 13, “Utility Programs”.

The Windows Menu

The **Windows** menu provides a submenu for each kind of window open in MULTI, allowing you easy access to any of them.

The Help Menu

The following table lists the selections available from the Project Manager **Help** menu:

MULTI Project Manager Help (F1)
Opens online help for the Project Manager.
Manuals
Opens the Manuals submenu, which displays a list of manuals appropriate to your version of MULTI.
Bookmarks
Opens the Bookmarks submenu, which displays all the Help Viewer bookmarks you have created. Bookmarks function across manuals and MULTI sessions. Select a bookmark to display the bookmarked page in online help. Select Manage bookmarks to open a window that allows you to display bookmarked pages and rename, delete, or reorder bookmarks.
About MULTI Project Manager
Displays version information about MULTI.

License Info
Displays the active licenses for MULTI.
Troubleshooting Info
Launches the gbugrpt utility program, which collects information about your MULTI installation and allows you to append it to a bug report form that you can fill out and email to Green Hills technical support.

The Compiler Driver

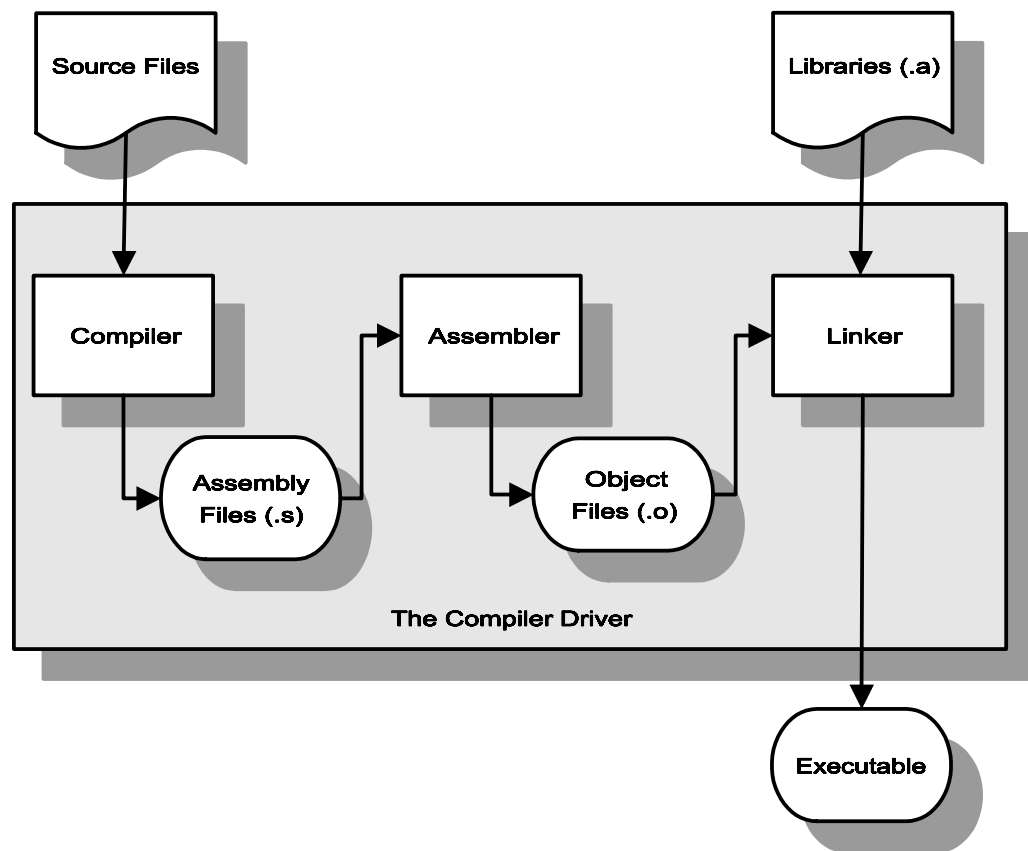
Chapter 4

This Chapter Contains:

- The Compiler Driver Syntax
- Building an Executable from C or C++ Source Files
- Working with Input Files
- Generating Other Output File Types
- Controlling Driver Information
- Using a Driver Options File
- Using Makefiles

The compiler driver is the command line wrapper for various toolchain components. The driver enables you to manually specify which source files you want to compile and/or when source files are necessary to link your program. This chapter explains how to use the driver and introduces some of the most important tasks it can perform. Like the Builder, it can control all the components of the Green Hills toolchain necessary to produce an executable from high-level source files:

- The optimizing compilers for C and C++, which compile high-level source files into PowerPC assembly language files.
- The **asppc** assembler, which translates PowerPC assembly files into object files.
- The **elxr** linker, which links object files (including object files in libraries) together to generate an executable.



In addition, the driver accepts many other forms of input files (see “Working with Input Files” on page 87) and can produce many different forms of output,

including libraries that the **ax** librarian creates by combining object files (see “Generating Other Output File Types” on page 90).

The Compiler Driver Syntax

The syntax for using the compiler driver is:

driver [*file* | *-option*]...

where:

- *driver* is one of the following:
 - **ccppc**: for C files
 - **cxppc**: for C++ files
- *file* is one or more of the following:
 - C or C++ source file
 - Assembly source file
 - Object file or library of object files
 - Linker directives file (see “Passing Linker Directives Files to the Driver” on page 88)
- *-option* is one or more compiler driver options. All of the options are case-sensitive (for example **-o** renames the output file, while **-Ogeneral** enables general optimizations), and most are host-independent. Many of the most commonly used options are discussed in this chapter. All of the driver options are listed and explained in Chapter 6, “Builder and Driver Options”.

Files and options are listed on the command line, separated by spaces.

The driver reads all options before processing any files. When two options represent different choices for the same feature, the later choice overrides the earlier one. If it encounters an unrecognized or invalid option, the driver will ignore it and issue a warning.

After reading the options, the driver processes files in the order they appear on the command line. If an error occurs in one file, processing will continue with the next file. If no errors occur, all object files and libraries will be linked together according to the order specified on the command line.

Building an Executable from C or C++ Source Files

When the driver receives source language files as input, it invokes the appropriate compiler, assembler, and linker to produce an executable.

To build an executable from a C source file called **hello.c**, use the C driver, **ccppc**, and enter the following:

```
ccppc hello.c
```

To build an executable from a C++ source file called **hello.cxx**, use the C++ driver, **cxppc**, and enter the following:

```
cxppc hello.cxx
```

In each case, an executable file called **a.out** is generated, together with the following additional files: **hello.o**, **a.map**, **a.dnm** and **a.dla**

The **hello.o** file is the object file which results from passing **hello.c** to the compiler and assembler. Whenever the driver generates an executable from a C, C++, or assembly language file, it retains the intermediate object files for future re-use. The **a.map** file is a map file produced by the linker. The **a.dnm** and **a.dla** files contain basic debugging information for use with the MULTI Debugger. For instructions on generating full debugging information for MULTI, see “Generating Debugging Information” on page 115.

To build an executable from multiple source files, list the files on the command line, separated by spaces, as follows:

```
ccppc hello.c foo.c bar.c
```

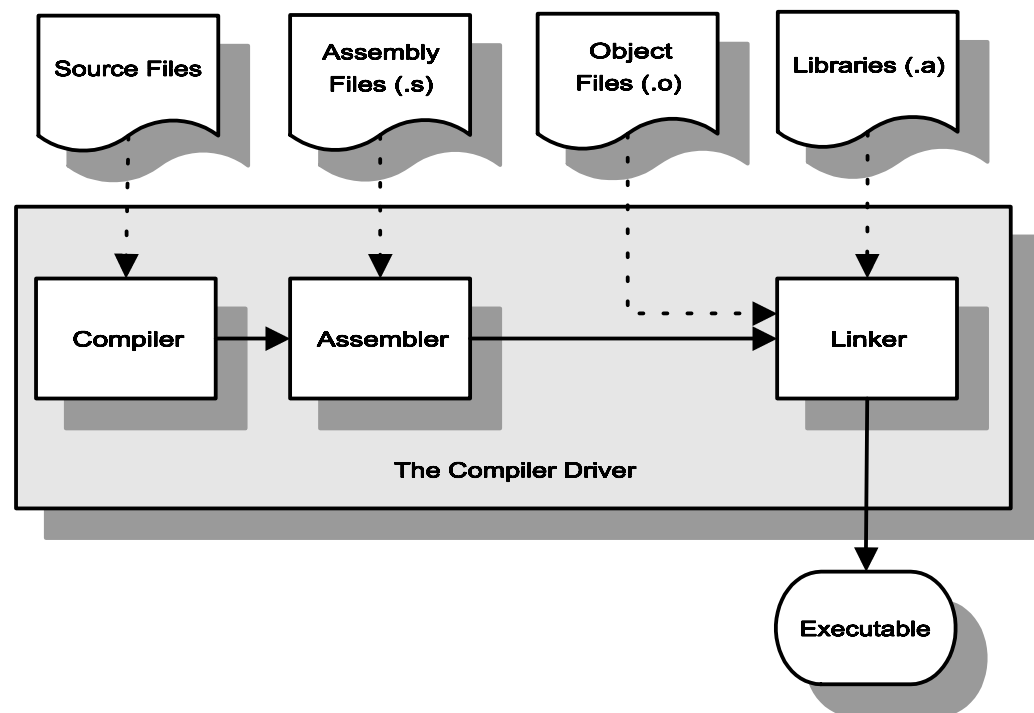
This command generates an executable called **a.out**, the debugging files **a.dnm** and **a.dla**, a map file **a.map**, and three object files, **hello.o**, **foo.o**, and **bar.o**.

Working with Input Files



Note The example command lines in this section, and in the remainder of this chapter, are shown using the C driver, **ccppc**. They will also work with the other drivers.

In addition to high-level language source files, the driver can also process assembly language files, object files, and libraries. The diagram below shows how these files are passed to the various tools in the toolchain.



Recognized Input File Types

The compiler driver determines the type of a file (and hence the compilation steps to perform on it) by reading its extension. For example, a file with a **.c** extension is a C source file and needs to be compiled with the appropriate language compiler, assembled, and then linked.

The following is a partial list of file extensions which the driver recognizes:

- C source files: **.c**, **.i**,
- C++ source files: **.C**, **.cc**, **.cpp**, **.cxx**

- Assembly language files: **.s**, **.asm**
- Object files: **.o**
- Library files: **.a**
- Linker directives files: **.ld**

Manually Setting Input File Types

On the command line, type **-filetype.suffix** to indicate that the next file on the command line has the type appropriate for *suffix*. For example:

```
ccppc -filetype.cc hello.i
```

compiles **hello.i** as if it were a C++ file.

To process all C files as C++ files, use the following syntax:

```
cxppc -dotciscxx files
```

Passing Multiple Input File Types to the Driver

To pass assembly files, object files, and libraries to the driver, add them to the command line, separated by spaces. For example:

```
ccppc hello.c foo.c bar.s baz.o libfoo.a
```

Source files of different languages can be included within the same executable by using the **Advanced→Project Options→Input Languages** or **-language** options. For more information, see “Project Options” on page 273.

Passing Linker Directives Files to the Driver

Linker Directives (**.ld**) files define the program sections of the executable and assign them to specific regions of memory. Several default **.ld** files are provided (see “Working with Linker Directives Files” on page 63), and the most appropriate one is used automatically.

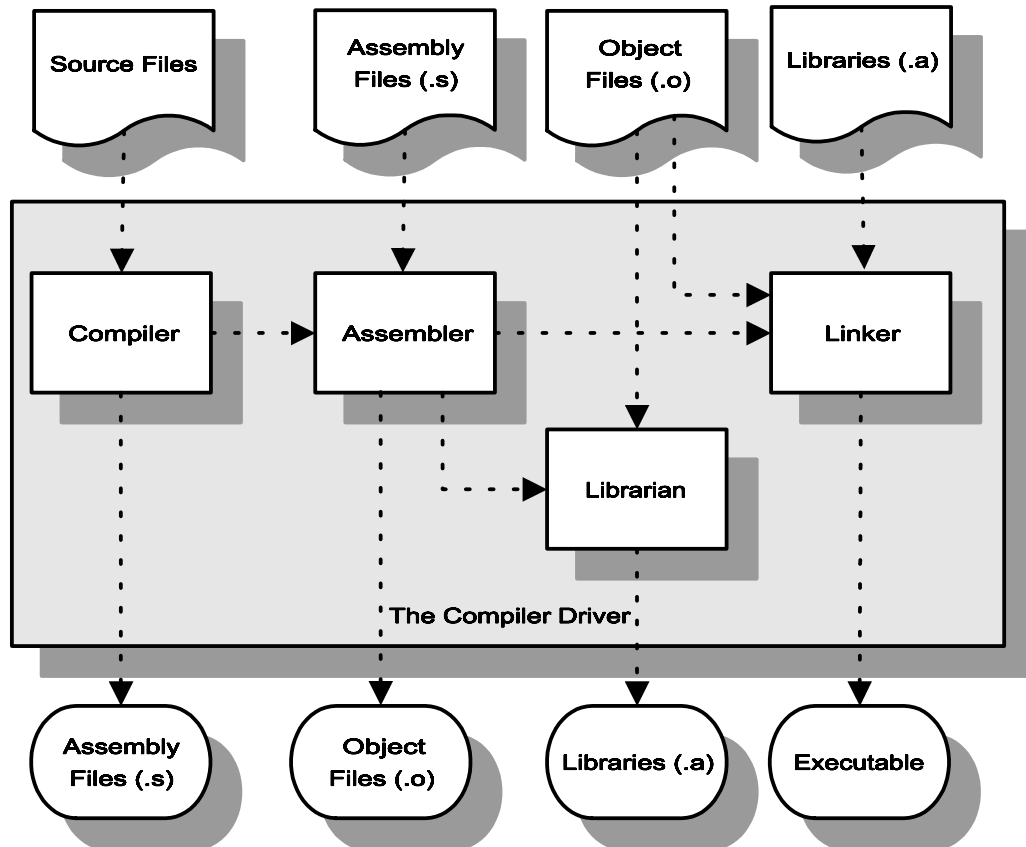
You can edit the default files to specify your own layout. For full documentation of the syntax, see “Configuring the Linker with Linker Directives Files” on page 437).

To pass a linker directives file to the driver, add it to the command line as follows (no option is required; the file is recognized by its extension):

```
ccppc hello.c mylinkfile.ld
```

Generating Other Output File Types

The driver can be instructed to halt the compilation process at various stages in order to produce assembly files, object files, libraries, or other forms of output.



To instruct the compiler to halt the build process after the compiler has generated assembly files, use the **-S** option as follows:

```
ccppc hello.c -S
```

This command produces an assembly language file called **hello.s**.

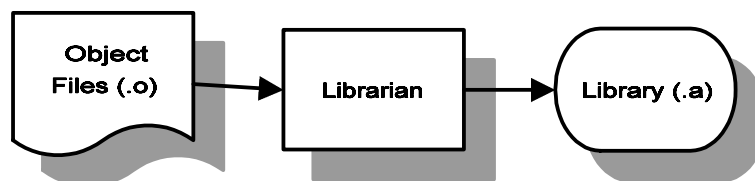
To instruct the compiler to halt the build process after the compiler has generated assembly files and the assembler has translated them to object files, use the **-c** option as follows:

```
ccppc hello.c -c
```

This produces an object file called **hello.o**.

Creating Libraries

You can pass high-level source files, assembly language files, and object files to the driver, and instruct it to collect them into a library. The driver invokes the compiler and/or the assembler (if required) to produce object files, and then invokes the librarian to combine these object files into a library.



To instruct the driver to create a library from the C source file **hello.c** and the object file **foo.o**, use the **-archive** option as follows:

```
ccppc hello.c foo.o -archive -o libfoo.a -G
```

This command produces a library of object files called **libfoo.a**, which contains two object files, **hello.o** and **foo.o**.



Note When using the **-archive** option to create a library, you must use the **-o** option to specify a name for it. The **-G** option is optional, but should be passed if you intend to use the MULTI Debugger.

Adding and Updating Files in Libraries

To add an object file to an existing library or to update an object file already included in a library, use the **-archive** and **-o** options in the same way you would to create that library. For example, to generate an object file from the C source file **hello.c** and add this object file to library **libfoo.a**, enter the following:

```
ccppc hello.c -archive -o libfoo.a
```

To perform other operations on libraries, you must invoke **ax** directly. For more information, see Chapter 12, “The ax Librarian”.

Driver Options for Intermediate Forms of Output

The following table lists all of the driver options that instruct the driver to stop at particular stages of the compilation process.



Note These options only apply to the compiler driver and cannot be set from the Project Manager.

-archive
Runs the ax librarian to generate a library (instead of running the linker to generate an executable program). For more information, see “Creating Libraries” on page 91.
-c
Produces an object file (called input-file.o) for each source file.
-E
Runs the preprocessor and writes the preprocessed file output to <code>stdout</code> unless the -o option is specified. This option can be useful when debugging preprocessor macros and <code>#include</code> files.
-Make
Prints to <code>stdout</code> makefile dependency rules for each object file. No other aspects of compilation are performed. Most error messages are suppressed. The dependency information will be wrong if the object file has a different name than the source or has an unexpected suffix. See also the -MD and -H driver options in “Generating Dependency and Header File Information” on page 100.
-merge_archive
Runs the ax librarian with the C and m options to generate a merged library instead of running the linker to generate an executable program.
-P
Similar to -E , but writes the output to input-file.i by default.
-Q
Produces only an inline file (called input-file.inf) for each source file. These files contain partially compiled code for functions generated during the first pass of the inliner, together with the list of the include files that the source file depends on (which are used by automatic dependency checking).
-S
Produces an assembly file (called input-file.s) for each source file.
-syntax
Checks the syntax without generating code.



Note See also the driver options associated with the **Linker→Generate Additional Output** option in “Linker Options” on page 244.

Output File Types

The following table lists the types of files generated by the MULTI Project Manager and Debugger:

.ael
A file generated by intex for an INTEGRITY application. It contains information that the Debugger uses to debug your application. If this file is deleted, you must relink the corresponding INTEGRITY application to regenerate it.
bmon.out gmon.out mon.out
Profiling data files that are read and written only by the toolchain.
.d
Source code dependency files generated by the compiler.
.dep
Program dependency files generated by the compiler.
.dba .dbo .dla .dlo .dnm
Debugging information files that are written only by the toolchain, and read by the toolchain and debugger. See “Debugging Options” on page 187.
.graph
Call graph information that is written only by the toolchain, and read by the user. See “Linker Output Analysis” on page 255.
.idep
Dependency files generated by Intex .
.ii .ti
C++ template information files that are read and written only by the toolchain.
.inf
Database files used for intermodule inlining that are read and written only by the toolchain.

.ipf
Database files used for interprocedural analysis that are read and written only by the toolchain.
.lcp
INTEGRITY-only files that are generated by the driver for Intex . It contains the command line used when building with -kernel so that Intex is able to relink the executable itself.
.lst
An assembler source listing that is written by the toolchain, and read only by the user. See “Assembler Options” on page 241
.map
A map file generated by the linker that contains information about how the program is laid out in memory. .map files are written by the toolchain, and read only by the user. See “Linker Output Analysis” on page 255.
.mem
An additional copy of the executable image that has been translated by the gmemfile utility. .mem is written by the toolchain, and read by non-GHS utility programs. See “Linker Options” on page 244.
.run
An additional copy of the executable image that has been translated by the gsrec utility. .run files are written by the toolchain, and read by non-GHS utility programs. See “Linker Options” on page 244.
.time
Files that are generated by the Project Manager when the file being processed does not generate an output file. This allows the Project Manager to track if and when the item was last processed for dependency tracking. .time is read and written only by the toolchain.

Controlling Driver Information

The following options control the transfer of information to and from the driver itself.



Note These options only apply to the compiler driver and cannot be set from the Project Manager.

@file
Instructs the driver to read command line arguments from <i>file</i> .
#
Displays the command lines generated by the driver (which are used to call the compiler, assembler, and/or linker for processing the input files) without executing them. For compatibility purposes, this option can also be entered as -dryrun or --driver_debug .
-help
Displays a list of some of the more commonly used options with brief descriptions.
-v
Displays the command lines generated by the driver as it calls the compiler, assembler, and/or linker for processing of input files.

Using a Driver Options File

When using multiple driver options, you may find that your command lines become overly long and hard to read. You can, instead, enter your options into a plain text file, which you pass to the driver using the *@file* option.

All characters in the file other than spaces, tabs, newline characters, and double quotes are literal.

Options and arguments must be separated by spaces, tabs, or newline characters. The double quote (") may be used to include spaces or tabs in a single argument as follows:

- If the argument begins with a ", then everything on the line up to the next " will be part of the argument, and those quotes will be discarded.

- Otherwise, if a " is in the middle of an argument, the " will not be discarded, but will cause any whitespace on the line up to the next " to be included in the argument.



Note Due to limitations in the command line parser, the **-bsp** and **-layout** options cannot be entered in a driver options file and must be placed directly on the driver command line, either before or after any *@file* option.

Example 1. How to Use a Driver Options File

The plain text file **myoptfile** contains the following information:

```
hello.c foo.c bar.c
-G
-Ospeed -OI
-o foo
```

To invoke the driver with the options file contained in **myoptfile**, enter:

```
ccppc @myoptfile -bsp=ipengine -layout=romrun
```

This command instructs the driver to read **myoptfile** and to:

- Pass files **hello.c**, **foo.c**, and **bar.c** to the compiler, assembler, and linker.
- Generate MULTI debugging information (**-G**).
- Compile the files in such a way as to maximize the speed of the final executable, including using two-pass inlining (**-Ospeed -OI**).
- Name the output executable **foo** (**-o foo**).
- Generate code tailored to the instruction set associated with the Bright Star Engineering ipEngine board, and assign the program to memory regions of that board using a board-specific linker directives files (**-bsp=ipengine -layout=romrun**).



Note You cannot include comments in a command file that you pass to the compiler driver with this option. Do not use this option to read a file containing assembler or linker options. Instead, use **-asmcmd=file** (see “Assembler Options” on page 241) or **-lnkcmd=file** (“Linker Options” on page 244) to pass the file option directly to the assembler or the linker, respectively. Note that linker directives files should be passed to the linker by using a known suffix, such as **.ld** or **.lnk**, rather than using this option.

Using Makefiles

The Green Hills compiler drivers are completely compatible with makefiles. This section discusses the basic issues involved in porting existing makefiles for use with Green Hills compilers.

In each of the following examples, we port a makefile from the fictional compiler **qcc** to the Green Hills compiler **ccppc**. We assume that **qcc** uses fairly standard command line syntax.

Example 2. A Very Simple Build

The following is an example of a very simple **qcc** makefile:

```
scanprog: scanprog.c
qcc -O2 scanprog.c -o scanprog
```

To convert this makefile for use with **ccppc**, you would need to make the following changes:

- Change the invocation of **qcc** to **ccppc**.
- Change the optimization option **-O2** to a Green Hills option, such as **-Ogeneral** (which enables general optimizations).

The resulting **ccppc** makefile will look like the following:

```
scanprog: scanprog.c
ccppc -Ogeneral scanprog.c -o scanprog
```

Example 3. Makefile Macros, Incremental Build

The following is a common format for a makefile. It uses an incremental build process (building `.c` files to `.o` files, and then linking them). It also uses makefile macros (such as `CC` and `CFLAGS`) to gather commonly-modified parts of the makefile in one place:

```
CC=gcc
CFLAGS=-g -I../include -DITERATIONS=3 -O3 -fprocessor=A
LFLAGS=-L../lib -lxyzlib -fprocessor=A
OFILES=bigprog.o main.o utils.o

bigprog: $(OFILES)
    $(CC) $(LFLAGS) $(OFILES) -o $@

bigprog.o: bigprog.c bigprog.h utils.h
main.o: main.c bigprog.h
utils.o: utils.c utils.h
```

To convert this makefile for use with `ccppc`, you would need to change the macros as follows:

- Change:

```
CC=gcc
```

to:

```
CC=ccppc
```

- Change:

```
CFLAGS=-g -I../include -DITERATIONS=3 -O3
```

to:

```
CFLAGS=-G -I../include -DITERATIONS=3 -Ospeed -OI
```


Example 4. Mixing Languages

In this example, two C files, one PowerPC assembly file, and one C++ file are linked together into a single executable:

```
OFILES=myprog.o utils1.o utils2.o interface.o

myprog: $(OFILES)
    gcc $(OFILES) -o myprog

myprog.o: myprog.c utils.h
    gcc -c myprog.c

utils1.o: utils1.c utils.h
    gcc -c utils1.c

utils2.o: utils1.s
    qasm -c utils1.s

interface.o: interface.cpp
    qc++ -c interface.cpp
```

Porting this example to use **cxppc** is easy because the Green Hills drivers recognize filename extensions and invoke the correct compiler or assembler automatically. Consequently, you would simply need to change all references to **gcc**, **qasm**, and **qc++** to **cxppc**, so that the makefile would read as follows:

```
OFILES=myprog.o utils1.o utils2.o interface.o

myprog: $(OFILES)
    cxppc $(OFILES) -o myprog

myprog.o: myprog.c utils.h
    cxppc -c myprog.c

utils1.o: utils1.c utils.h
    cxppc -c utils1.c

utils2.o: utils1.s
    cxppc -c utils1.s

interface.o: interface.cpp
    cxppc -c interface.cpp
```

Green Hills Equivalents to GNU Tools

The following table provides the Green Hills equivalents to common GNU tools:

GNU	Green Hills
CC=gcc	CC=ccppc
CXX=g++	CXX=cxppc
LD=ld	LD=cxppc
AR=ar	AR=ax AR=cxppc -archive

Use AR=ax for libraries written entirely in C, and for which you do not need debug information. For C++ libraries involving instantiable entities such as templates, either compile using the **--link_once_templates** option, or use AR=cxppc -archive, which requires additional modifications to the rules in your makefile that create libraries.

Generating Dependency and Header File Information

These options provide information about your makefile structure:

-H

Prints to `stderr` a list of files opened by `#include` directives during normal compilation. Files are compiled and linked normally. This option can be used together with **-syntax** to print the names of header files without producing any other output.

-MD

Produces the following makefile dependency files:

- One ***input-file.d*** for each object file.
- One ***input-file.dep*** for each program.

Duplicate dependencies are eliminated and only one rule is output for a single object file. The source and program dependency files have different extensions to prevent one file from overwriting another in the case that an object file has the same name as a program file.

The backslash (`\`) character is used to form continuation lines.

This option has no effect if **-syntax** is passed.

This option only applies to the compiler driver and cannot be set from the Project Manager.

-MMD

Behaves the same as **-MD**, but does not include system header directories (see **-sys_include_directory**) or C++ headers in the output.

This option only applies to the compiler driver and cannot be set from the Project Manager.

Developing for PowerPC

Chapter 5

This Chapter Contains:

- PowerPC Characteristics
- Structure Packing
- Specifying a PowerPC Target
- Generating Debugging Information
- Using Your Own Header Files and Libraries
- Controlling the Assembler
- Controlling the Linker
- Text and Data Placement
- Customizing the Green Hills Run-Time Environment
- Other Topics

This chapter describes the PowerPC target environment and provides information about some of the Builder and driver options that access and control features of that environment.

The Green Hills PowerPC compiler, assembler, and linker comply with the *PowerPC Embedded Application Binary Interface* specification (*PowerPC EABI*).

PowerPC Characteristics

The PowerPC memory is byte addressed with 32-bit addresses. Bits are numbered with bit zero as the most significant bit.

By default, bytes are ordered in the big endian format, with the most significant byte of a multiple-byte value stored at the lowest address:

Address 0	Address 1	Address 2	Address 3
Most Significant Byte			Least Significant Byte

The reverse byte order, little endian, is also supported for most processors.

Floating-point values use IEEE-754 format (32 and 64 bits) and are in the same byte order as other values.



Note Although we follow the IEEE-754 floating-point format, floating-point operations might not generate the exact values required by the specification.

On systems where the compiler generates code for a floating-point unit, the floating-point computation is as accurate as the underlying hardware (Green Hills does not supply software libraries to handle cases that the hardware may ignore or signal an exception on). On systems where the Green Hills floating-point emulation routines are used instead of hardware, the floating-point emulation routines are designed for speed, and the results may differ from the IEEE specification, particularly in exceptional cases such as NaNs and denormals.

In cases where the compiler uses the `fsel` floating-point conditional move instruction to generate code for floating-point comparisons, the results might not be accurate for comparisons involving NaNs and infinities of the same sign. Use the `-no_use_fsel` option to disallow use of the `fsel` instruction. For

a description of these inaccuracies, see Book I, Appendix E.3.4 of the *Power Instruction Set Architecture, Version 2.05*.

Character encoding is ASCII.

In C and C++, bitfields are allocated starting at the lowest memory address. Bitfields begin on the most significant bit in big endian mode, and on the least significant bit in little endian mode. Each structure, union, and array is aligned to the maximum alignment requirement of any of its components.

Register Usage

The registers for standard PowerPC processors are listed in the tables below. *Volatile registers* can have their contents destroyed by subroutine calls, while *non-volatile registers* will not.

The general purpose registers are:

Register Name	Usage
r0	Volatile (scratch)
r1	Stack Pointer (<i>sp</i>)
r2	Read-Only SDA pointer
r3-r10	Parameter registers (volatile); r3 is also the return register
r11-r12	Volatile
r13	Read-write SDA pointer
r14-r26	Non-volatile
r27	Non-volatile or frame pointer when needed and 64-bit software floating point support enabled
r28	Non-volatile
r29	Non-volatile or frame pointer when needed and 64-bit software floating point support disabled
r30	Non-volatile or Static Link Register
r31	Non-volatile

The floating-point registers are:

Register Name	Usage
f0	Volatile
f1-f8	Parameter registers (volatile); f1 is also the return register
f9-f13	Volatile
f14-f31	Non-volatile

In compliance with the PowerPC EABI, **crt0.ppc** initializes r13 to `__ghsbegin_sdabase + 0x8000` and r2 to `__ghsbegin_sdata2 + 0x8000` (the start of the appropriate read-only small data area section plus 32K). For more information about the SDA, see “Special Data Area Optimizations” on page 137. These registers remain fixed during the execution of an EABI-compliant application.

Data Type Sizes and Alignment Requirements

This section lists the data type sizes and alignment requirements for C and C++. The *size* of a data type is the number of bits the type requires. The *alignment requirement* of a data type tells the compiler how to control the type’s starting address in memory.

An instance of a data type is *aligned* if the compiler places it at an address offset $a*n$ bits from the beginning of the section, where a is the type’s alignment requirement, and n is an integer greater than or equal to 0. A data type’s size is always a multiple of its alignment requirement, so that multiple instances of the data type remain aligned when the compiler places them in an array.

See the *PowerPC EABI* and related documents for more information about size and alignment requirement.

C and C++ Data Types

C/C++ Data Type	Size (bits)	Alignment Requirement (bits)
char	8	8
short	16	16
int	32	32
long	32	32
float	32	32

C/C++ Data Type	Size (bits)	Alignment Requirement (bits)
double	64	64
long long	64	64
long double	64	64
vector	128	128
* (pointer)	32	32
enum (default)	32	32



Note To reduce the size and alignment requirement of the enum type to 8 or 16 bits where possible:



Set the **C/C++ Compiler**→**Data Types**→**Use Smallest Type Possible for Enum** option to **On** (`--short_enum`). For more information, see “**Use Smallest Type Possible for Enum**” on page 226.

Calling Conventions

Argument promotion rules specific to the programming language are applied.

Subroutine call arguments and return values are assigned to registers and the stack according to the rules in the *PowerPC EABI*. For object files built with PIC, the EABI is extended to use a few additional relocations. Generally, scalar quantities are passed in the parameter registers if possible, otherwise they are passed on the stack. If a structure is passed by value, then a local copy is made by the caller routine. Scalar values return in `r3` or `f1`. Structures and integral values up to 64 bits in size are returned in `r3` and `r4`; larger structure return values are copied to an address passed by the caller as a “hidden” first argument.

Most of the time, a call to a function uses a `bl` instruction that saves the return address in the system register `lr`. Most of the time, return uses a `blr` instruction.

Accesses to parameters or local stack storage are normally relative to the stack pointer `r1`. Functions that call the dynamic stack space allocation routine **alloca** have their parameters and local stack storage accessed via frame pointer (see “Register Usage” on page 105).

Structure Packing

This section discusses structure packing and shows you how to specify structure packing options to the compiler. In general, you use the `#pragma pack (n)` directive to instruct the compiler to pack all instances of a particular structure type. You then use the `__packed` type qualifier when declaring any pointer that might point to a field in a packed structure. This section is organized into several parts that contain more detailed information about how structure packing works:

- “Understanding Structure Packing” on page 108
- “Using `#pragma pack` to Pack All Instances of a Structure Type” on page 110
- “Using the Packing Builder Option to Pack All Structures” on page 111
- “Pointing to Packed Structures with the `__packed` Type Qualifier” on page 111

Understanding Structure Packing

Structure packing is a feature you can use to reduce the amount of memory that instances of your structures require. Understanding how structure packing works is important, because this feature affects the alignment of your structures and the fields they contain.

A structure is *aligned to n bytes* if the compiler places it at an address offset by a multiple of n bytes from the beginning of the section. By default, a structure’s alignment requirement is equal to the largest of any of its fields’ alignment requirements. A structure is *naturally aligned* if it is aligned to its alignment requirement. Otherwise, the structure is *misaligned*. Like all data types, a structure’s size is a multiple of its alignment requirement, so that multiple instances of the structure remain aligned when the compiler places them in an array.

The Green Hills compilers always allocate fields of a structure in the order specified in the declaration. A field is aligned to n bytes if the compiler places it at an address offset by a multiple of n bytes from the beginning of the containing (unpacked) structure. A field is naturally aligned if it is aligned to its alignment requirement. Otherwise, the field is misaligned. For a list of the alignment requirements for each data type, see “Data Type Sizes and Alignment Requirements” on page 106.

If the compiler is placing a field or structure, and placing it at the next available address would make it misaligned, the compiler will insert bytes of *padding* until it can place the field or structure so that it is naturally aligned.

Structure packing reduces the amount of padding the compiler inserts between fields in a structure. When using structure packing, you give the compiler a *packing alignment*. When the compiler places a field in a packed structure, it aligns that field to its alignment requirement, or the packing alignment, whichever is smaller. This means that the compiler will insert at most $n-1$ bytes of padding between fields, where n is the packing alignment. The compiler aligns a packed structure itself to the largest of any of its fields' alignment requirements, or the packing alignment, whichever is smaller.

As an example, say you define struct `s` like so:

```
struct s {
    short a;
    int b;
    char c;
} d[2];
```

The following diagram shows how the compiler will allocate `d[2]` using its natural alignment, with a 2-byte packing alignment, and with a 1-byte packing alignment:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Natural	d[0]												d[1]											
	a	a			b	b	b	b	c				a	a			b	b	b	b	c			
2-byte	d[0]								d[1]															
	a	a	b	b	b	b	c		a	a	b	b	b	b	c									
1-byte	d[0]								d[1]															
	a	a	b	b	b	b	c		a	a	b	b	b	b	c									

Notice that when the compiler uses structure packing, `d[2]` takes up fewer bytes, but some of the fields and structures are misaligned. Because some PowerPC processors do not support accesses to misaligned fields, the compiler generates additional code for those processors when fields are misaligned. This code is usually less efficient than code that accesses naturally aligned fields.

Using #pragma pack to Pack All Instances of a Structure Type

You can instruct the compiler to pack all instances of a particular structure type by using the `#pragma pack (n)` directive, where *n* is the packing alignment in bytes, is a power of two, and is less than or equal to `__ghs_max_pack_value`. The directive must appear before the beginning of the structure definition, *not* inside it. The directive affects all subsequent structure definitions until the next `#pragma pack (n)` directive, or the end of the translation unit. If you do not specify *n*, then the compiler does not pack subsequent structures, unless you have set the **C/C++ Compiler→Alignment & Packing→Packing (Maximum Structure Alignment)** Builder option (see “Using the Packing Builder Option to Pack All Structures” on page 111 for more information).

Example 1. #pragma pack Scope

When you use the `#pragma pack (n)` directive around a structure declaration, the compiler packs all instances of that structure. The directive does not affect instance declarations.

```
struct s {
    short a;
    int b;
    char c;
} j;
#pragma pack(2)
struct s k;
struct s2 {
    short a;
    int b;
    char c;
} x;
#pragma pack()
struct s2 y;
```

The size of both `j` and `k` is 12. The compiler will place two bytes of padding between field `a` and field `b`. Because you already defined `struct s`, the `#pragma pack (2)` directive does not affect the declaration of `k`.

The size of both `x` and `y` is 8. The compiler will not place padding between field `a` and field `b`. Because you already defined `struct s2`, the `#pragma pack ()` directive does not affect the declaration of `y`.

Using the Packing Builder Option to Pack All Structures

You can set the packing alignment for all structures in your program by using the **C/C++ Compiler→Alignment & Packing→Packing (Maximum Structure Alignment)** Builder option.



Set the **C/C++ Compiler→Alignment & Packing→Packing (Maximum Structure Alignment)** Builder option (**-pack=*n***), where *n* is the packing alignment in bytes, and has a value of 1, 2, 4, or 8.



Note When using this Builder option on a project, you must use the `__packed` qualifier on any pointer that points to a field in any structure declared in that project. See “Pointing to Packed Structures with the `__packed` Type Qualifier” on page 111 for more information.

Pointing to Packed Structures with the `__packed` Type Qualifier

The `__packed` type qualifier tells the compiler that a particular pointer might point to a misaligned structure or field. The compiler uses this information to generate additional code that handles them in software, if necessary.

Because any field in a packed structure might be misaligned, you must use the `__packed` type qualifier when declaring any pointer that points to a field in a packed structure. For instance, if you pack the following structure:

```
#pragma pack(2)
struct s {
    short a;
    int b;
    char c;
} x;
#pragma pack()
```

and then create a pointer to a field in that packed structure:

```
int *p = &(x.b);
```

Because `&(x.b)` points to a misaligned field in a packed structure, the compiler issues warning 1973, and the program might exhibit unexpected behavior. Instead, use:

```
__packed int *p = &(x.b);
```

The program now runs as expected, because the compiler generates code to handle misaligned accesses through `*p`.

If you take the address of the entire packed structure into a pointer of that structure's type, it is not necessary to use the `__packed` type qualifier, because the structure type is already declared as packed. For example:

```
/* Correct */
struct s *p1 = &x;

/* Correct, but __packed is unnecessary here*/
__packed struct s *p2 = &x;

char buf[sizeof(struct s)];
memcpy(buf, &x, sizeof(struct s));

/* Correct - p2 can point to a struct object with 1-byte alignment */
p2 = (__packed struct s *)buf;

/* Incorrect - buf might not have the 2-byte alignment required by struct s */
p1 = (struct s *)buf;
```



Note Do not declare objects as both `__packed` and `volatile` simultaneously, or put `volatile` fields inside a packed structure. Loads and stores of such objects might exhibit unexpected behavior.

Specifying a PowerPC Target

The Builder and compiler drivers support many of the most popular PowerPC boards and the entire PowerPC family of microprocessors. The compilers can produce more efficient code if you specify the target on which you intend to run the finished executable. Target options may affect the predefined preprocessor symbols, the compiler, the assembler, and the selection of header files, libraries, and linker directive files used by the linker.

Many of the most popular PowerPC boards are supported through tailored linker directives files that greatly simplify the process of linking and loading an executable. To specify a particular target board:



When creating a new project with the **Project Wizard** (see “Creating A New Project” on page 22), choose the appropriate target board. To subsequently change your target in the Project Manager, click **Edit** → **Set Build Target**, and select a target from the **Target Selector** dialog box.

To specify a target board with the driver, pass the **-bsp board** option. To obtain a list of supported boards, enter **-bsp=?** (or **-bsp=\?** in many shells).

The **-bsp=** option may imply other settings according to the requirements of your board. It also allows you to use the **-layout=** option. **-bsp=** and **-layout=** select the linker directive file used by the toolchain when building your project. For more information about the **-layout=** option, see “Assigning Program Sections to ROM and RAM” on page 132.

If your board is not supported, you should specify your target processor instead. Specifying a target processor enables the driver to determine the most efficient instruction set to use in code generation.

To specify a target processor:



When creating a project with the **Project Wizard** (see “Creating A New Project” on page 22), expand the **Generic** option in the **Board Name** box, and choose the appropriate processor. This corresponds to the **-bsp=generic** driver option. To subsequently change your target in the Project Manager, click **Edit** → **Set Build Target**, and choose a target from the **Target Selector** dialog box.

To specify a target processor with the driver, pass the **-cpu=cpu** option. To obtain a list of supported processors, enter **-cpu=?** (or **-cpu=\?** in many shells).

PowerPC Processor Variants

The following table lists the supported PowerPC processor variants. The default processor type is PowerPC Espresso.

All PowerPC cpu macros are also defined without the two trailing underscores.

Processor Name	Driver Option <code>-cpu=cpu</code>	Predefined Macro Name	Defaults	
			Library Directory	Floating Point
PowerPC Espresso	espresso	<code>__espresso__</code>	ppc	Hardware

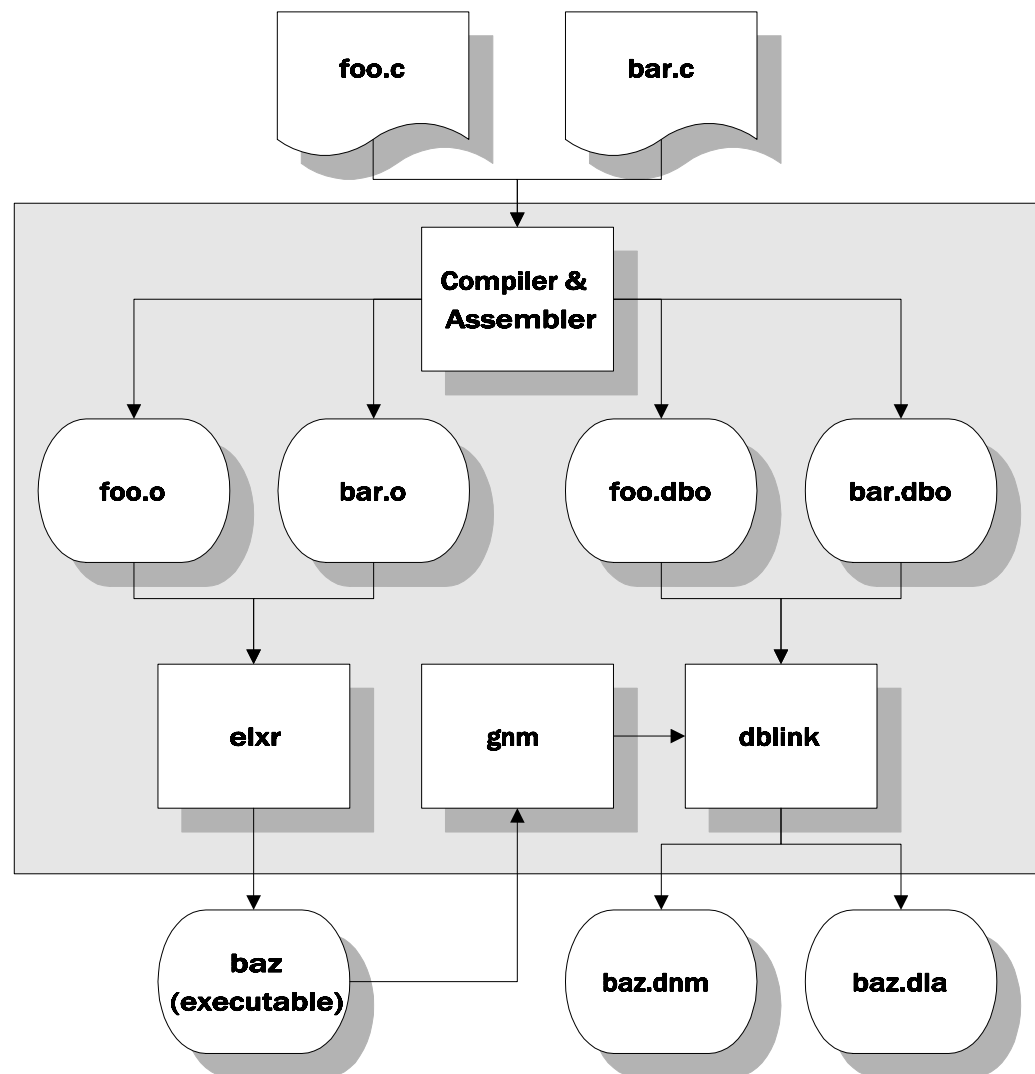
Generating Debugging Information

To generate debugging information for use with the MULTI Debugger:



Set the **Debugging**→**Debugging Level** option to **MULTI (-G)**.

The diagram below illustrates the creation of debugging information:



As the compiler and assembler translate each C or other high-level language file into an object file, they also generate associated debugging information. For each object file created, a separate file with the same name but a **.dbo** extension

is produced to hold this information. After the executable is linked together, these individual debugging information files are combined into a structure that can be quickly searched and incrementally loaded.

In the diagram, the files **foo.c** and **bar.c** are passed to the toolchain, which generates files **foo.o** and **bar.o** and links them together to generate an executable, **baz**. In addition, the debugging information files **foo.dbo** and **bar.dbo** are generated and passed to the **dblink** utility, which also extracts the executable's namelist information (via the **gnm** utility), and generates the following two files:

- **baz.dnm** — the debug symbol table for **baz**.
- **baz.dla** — an archive of the **.dbo** files associated with **baz**

When the executable is opened in the MULTI Debugger, **baz.dnm** controls the loading of debugging information from **baz.dla** as necessary.



Note

- The creation of debugging information for libraries is performed in the same way as for object files, except that an additional file with a **.dba** extension is generated to combine the debugging information for all of the members of the library.
- Debugging information is only created or updated when you create an executable or a library with the Builder or driver, and not when the linker or librarian are invoked separately.
- If you move an executable, library, or object file, you should also move the associated debugging information files. Otherwise, it may be necessary to rebuild or relink your program before you can use the MULTI Debugger.
- The files **foo.dnm** and **foo.dla** are generated even if the **Debugging→Debugging Level** option is not set to **MULTI (-G)**, so that MULTI can obtain skeletal debugging information for the executable. However, you must enable this option in order to perform source-level debugging and to view assembly language interlaced with your source.

For more information about debugging options, see “Debugging Options” on page 187. For information about using the MULTI Debugger, see *MULTI: Debugging*.

Obtaining Profiling Information

The MULTI Debugger can analyze various forms of profiling information to help you make your program run more efficiently. This section describes how to compile your program so that profiling information is available to the Debugger and the memory requirements for each supported profiling method. For information about the various forms of profiling, and how to use them, see Chapter 15, “Collecting and Viewing Profiling Data” in the *MULTI: Debugging* book.

Call Count Profiling

Call Count profiling (**-p**) causes the compiler to add instrumentation code to the beginning of every procedure in your program. The program will usually have a larger `.text` section with this type of profiling enabled, but will have a smaller `.text` section than with coverage profiling.

Call count profiling also dynamically allocates tables in heap memory proportional in size to the number of procedures actually executed while running your program. Call Count profiling uses an amount of heap memory equal to one pointer and one 32-bit integer counter (8 bytes on most systems) for each procedure in your program, plus some overhead for the linked lists it uses to hold the counters.

To obtain call count profiling information:



Set the **Debugging→Profiling - Call Count** option to:

- **On (-p)**, or
- **On with Call graph (-pg)**.

If you try to use call count profiling and you get the error message `"mcount : could not allocate memory"` from the target, there is not enough heap memory available to use call count profiling. You can increase the size of the heap by changing the size of the `.heap` section in the linker directives file.

Call count profiling is written to a file called **mon.out** when your program exits or when you issue the **profdump** command to the MULTI Debugger (see “Manually Dumping Profiling Data” in Chapter 15, “Collecting and Viewing Profiling Data” in the *MULTI: Debugging* book). You can usually find this file in the MULTI working directory, the directory of the simulator or debug server

that you last used to run your program, or the same directory as your program, depending on your configuration.

Call Graph Profiling

Call graph profiling (**-pg**), also known as **gcount**, causes the compiler to add instrumentation code to the beginning of every procedure in your program. This type of profiling creates a larger `.text` section when enabled, but will have a smaller `.text` section than with coverage profiling.

Call graph profiling requires an amount of heap memory that is, by default, between 1.5 and 2 times the size of the `.text` section. The memory is used to hold a hash table and a pool of (`from_pc`, `to_pc`, `count`) arcs, which forms a call graph with weighted edges. This pool is large enough to store the call graphs of most programs.

If you see the following message from the target, your program has run out of available heap memory:

```
gcount: could not allocate memory
```

If you see the following message from the target, your program has a complicated call graph that cannot be represented with the arc pool:

```
gcount: not enough arcs to record call graph
```

You can fix this by customizing the low-level system library **libsys.a** (see “Customizing the Run-Time Environment Libraries and Object Modules” on page 751) and following the instructions contained in the comments of **ind_gprf.c**.

Call graph profiling information is written to a file called **gmon.out** when your program exits or when you issue the **profdump** command to the MULTI Debugger (see “Manually Dumping Profiling Data” in Chapter 15, “Collecting and Viewing Profiling Data” in the *MULTI: Debugging* book). You can usually find this file in the MULTI working directory, the directory of the simulator or debug server that you last used to run your program, or the same directory as your program, depending on your configuration.

Coverage Profiling

Coverage Profiling (**-a**), also known as *Block Count* or **bcount**, causes the compiler to add instrumentation code to most basic blocks in every procedure built with this option, so the program may have larger `.text` and `.data` sections when this option is enabled. It also causes the compiler to add static tables to the data section for each procedure proportional in size to the size of the procedure. A typical proportion might be one-fifth or one-sixth.

To obtain coverage profiling information:



Set the **Debugging**→**Profiling - Coverage** option to **On (-a)**.

Call coverage profile information is written to a file called **bmon.out** when your program exits or when you issue the **profdump** command to the MULTI Debugger (see “Manually Dumping Profiling Data” in Chapter 15, “Collecting and Viewing Profiling Data” in the *MULTI: Debugging* book). You can usually find this file in the MULTI working directory, the directory of the simulator or debug server that you last used to run your program, or the same directory as your program, depending on your configuration.



Note Coverage profiling might not be reliable when used in conjunction with compiler optimizations, link-time optimizations (such as constant propagation), and options that cause code instrumentation (such as run-time error checking). INTEGRITY projects have link-time constant propagation (**--link_constprop**) enabled by default. If you are using INTEGRITY 10, pass the **--no_link_constprop** option when using coverage profiling. If you are using INTEGRITY 5, constant propagation cannot be disabled.

Target-Based Timing Profiling

This section describes how to obtain *Timing Profiling* information for stand-alone and u-velOSity targets.

Target-Based Timing Profiling (-timer_profile) also known as **manprf**, is supported by **ind_manprf.c** and **ppc_manprf.h** in the source of **libsys.a**, and parts of the implementation might also be in the Board Initialization library **libboardinit.a** if your board contains that optional library. See “Board Initialization Library libboardinit.a” on page 760 for a description of this interface and the **board_info.txt** file in your project, if one exists. This form of profiling, which is generated by the target periodically sampling its own program counter, is faster and more accurate than host-driven profiling.

Target-based timing profiling requires a statically-allocated memory buffer, whose size is adjustable in *install_dir/src/libsys/ind_manprf.c*. The default size is 480 bytes. This buffer is used to hold the program counter samples before they are sent to MULTI. The buffer is statically allocated, so it will typically be allocated to `.bss`, not `.heap`. Rather than being written to a file by the target, target-based timing profiling information is intercepted by the debug server when the memory buffer fills.

The Target-Driven Timer Profiling library module, **ind_manprf.o**, is not linked into your program by default. To link with it and enable this form of profiling, follow these steps:

1. Right-click the program’s project file and select **Configure**. In the dialog box that opens, select the **Board Initialization** check box (if present) and click **Finish**.
2. Review **ind_manprf.c**, the **ppc_manprf.h** header file, and (if present) your board’s **board_info.txt** file for instructions. If your board is not explicitly supported, you must provide an implementation using the existing code as an example.
3. Rebuild the system library **libsys.a** and (if available for your board) the board initialization library **libboardinit.a** (see “Board Initialization Library libboardinit.a” on page 760 for a description of this interface).
4. To link with the **ind_manprf.o** module:



Set the **Debugging**→**Profiling - Target-Based Timing** option to **On** (-timer_profile).

Rebuild your application. If linker errors result, you may need to review the settings for your board in the Target-Driven Timing Profiling source files mentioned in the preceding.

5. Open your application in the debugger, enable profiling and run your program in the usual way. For more information about profiling, see Chapter 15, “Collecting and Viewing Profiling Data” in the *MULTI: Debugging* book.

Limitations and Suggestions

Target-based timing profiling gathers stochastic samples of the target program’s location at the time of periodic timer interrupts. For best results, let your program run for a long time to gather a large number of profiling data samples.

This method of gathering samples can generate misleading results if the sampling timer also causes other program actions, or if the program is dealing with external events at or near a multiple of the sampling timer’s frequency.

Using Your Own Header Files and Libraries



Note For information about the headers and libraries provided with your distribution, see Chapter 17, “Libraries and Header Files”.

You might want to create your own libraries of frequently used functions and instruct the compiler to link against them. While the compiler knows where to find the standard libraries and header files provided with your distribution and links against them, you must tell it where it should search for your own headers and libraries. At link-time, the linker will attempt to resolve external references in the source files you pass to it by linking against your libraries before searching the standard libraries.

Instructing the Compiler to Search for Your Headers

To instruct the compiler to look for header files in a new directory:



Enter the directory in the **Project→Include Directories** option (**-Idirectory**).

You can specify multiple directories. The compiler searches the directories in the order in which they are specified.

Using Two Forms of the Include Directive

The compiler recognizes both standard forms of the `#include` directive (that is, `#include <header>` or `#include "header"`). These forms can specify either a full path or only a filename. The full range of possible types of `#include` directives are identified in the following:

	Linux/Solaris host	Windows host
A	<code>#include "/src/h/header.h"</code>	<code>#include "f:\src\h\header.h"</code>
B	<code>#include </src/h/header.h></code>	<code>#include <f:\src\h\header.h></code>
C	<code>#include "header.h"</code>	<code>#include "header.h"</code>
D	<code>#include <header.h></code>	<code>#include <header.h></code>

The compiler searches for header files included with the various types of `#include` directive in the following ways:

- If the `#include` directive specifies a full path to the header file (cases A and B above), the compiler searches in that location only. It does not search for another version of the file in any other directory.
- If the `#include` directive specifies only a filename or a partial path for the header file, and uses quotes (case C above), the compiler searches in the following directories, in order:
 1. The directory of the source file that contains the `#include` directive.
 2. Directories specified with the **Project→Include Directories** option (**-Idirectory**), in the order in which they appear on the command line.
 3. The default directories (see “C and C++ Header File Directories” on page 705).
- If the `#include` directive specifies only a filename or a partial path for the header file, and uses angle brackets (case D above), the compiler searches in the following directories, in order:
 1. Directories specified with the **Project→Include Directories** option (**-Idirectory**), in the order in which they appear on the command line, and then
 2. The default directories (see “C and C++ Header File Directories” on page 705).

Modifying Header File Searches with the **-I** Option

Specifying a dash (–) in the **Project→Include Directories** list (**-I**) changes the rules for searching for header files where only a filename or partial path is given (cases C and D):

- For the directive `#include "header.h"` (case C in the previous section), the compiler searches in the following directories, in order:
 1. Directories specified with **-Idirectory**, in the order in which they appear on the command line.
 2. The default directories.

The compiler does not search in the directory of the source file, unless the **--search_source_directory** option is used.

- For the directive `#include <header.h>` (case D in the previous section), the compiler searches in the following directories, in order:
 1. Directories specified with `-Idirectory` that appear on the command line after the `-I-` option.
 2. The default directories.

For example, if the command line includes:

```
-Iquote -I- -Iangle
```

Then `#include "foo.h"` would refer to the first matching file of **quote/foo.h** or **angle/foo.h**, but `#include <foo.h>` could refer only to **angle/foo.h**.

Modifying C++ Header File Searches

Since C++ headers do not necessarily have a **.h** extension, there may be confusion between the legitimate headers and other files or subdirectories with the same name. To limit such confusion, you can create a new directory, dedicated to C++ headers without extensions, and specify this directory to the compiler for searching as follows:



Specify the appropriate directory in the **Advanced→Preprocessor Options→Advanced Include Directories→C++ Include Directories** option (`--cxx_include_directory directory`).

When searching for C++ headers without extensions, the driver ignores any directories specified with the **Project→Include Directories** option (`-Iinclude_directory`), but it does search in the directory specified with this option.

Instructing the Compiler to Link Against Your Libraries

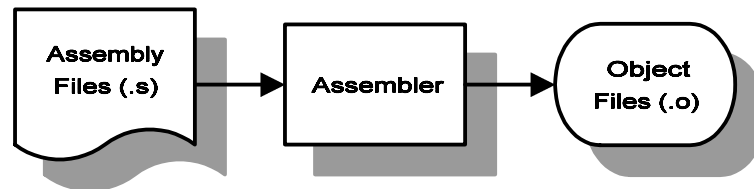
To instruct the compiler to link against your libraries:



- Specify the directories to search in the **Project→Library Directories** option (**-Llibrary_directory**).
- Specify the library names to link against in the **Project→Libraries** option (**-llibrary**). Note that the names of libraries must be specified in their abbreviated form. For example, a library called **libfoo.a** would be specified as **foo**.

Controlling the Assembler

Unless you instruct them otherwise, the Builder or driver automatically invokes the assembler, **asppc**, to process the assembly file output of the compiler.



You can control the most commonly used features of the assembler with Builder and driver options. For example, to instruct the assembler to produce a source listing file:

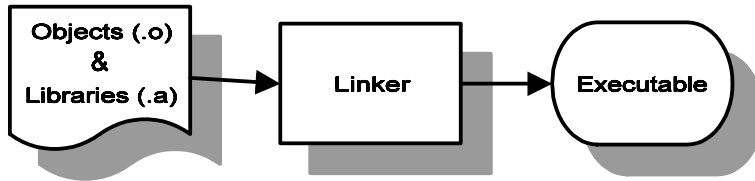


Specify a filename in the **Assembler→Source Listing Generation** option (**-list[=file]**).

For a list of all the assembler-specific Builder and driver options, see “Assembler Options” on page 241. For detailed documentation of the assembler, see Chapter 9, “The asppc Assembler”.

Controlling the Linker

Unless you instruct them otherwise, the Builder or driver automatically invokes the linker, **elxr**, to combine the object file output of the assembler.



You can control the most commonly used features of the linker with Builder and driver options. For example, to delete unused functions:

	Set the Linker→Linker Optimizations→Deletion of Unused Functions option to On (-delete) .
---	---

For a list of all the linker-specific Builder and driver options, see “Linker Options” on page 244. For detailed documentation of the linker, see Chapter 11, “The elxr Linker”.

Text and Data Placement

The compiler assigns all text and data to *program sections*, which are named collections of objects. During the link step, the linker collects all data for each program section and allocates it in memory, guided by a section map defined in your linker directives (**.ld**) file. The section map specifies the location of each program section in the final output file. Normally, the compiler and linker use a default set of program sections, but you can also define custom program sections, assign specific data to them, and assign them to specific memory blocks on your target.

The following documentation explains how the compiler assigns data to default program sections and how to create and use custom sections. It goes on to cover additional topics that explain how to manipulate the placement of text and data (such as in registers or special data areas), and how to access it.

Default Program Sections

The default program sections for text and data (except local automatic variables) are:

- `.text` — which holds program code.
- `.data` — which holds variables with explicitly initialized values (e.g. `int i=1;`).
- `.bss` — which holds variables that are not explicitly initialized and that are not COMMON variables (for example, `static int j;`). The Green Hills startup code initializes this section to all zeros, as required by the C and C++ languages.
- `.rodata` — which holds some read-only data, such as `const` variables. This data varies depending on your target.
- `COMMON` — which holds global variables in C that are not explicitly initialized and whose modules were not compiled with the `--no_commons` option (see “**Allocation of Uninitialized Global Variables**” on page 227). When linking an executable, the linker reassigns each variable in this section based on whether or not it is explicitly initialized in any of the object files. By default, the linker reassigns uninitialized global variables in this section to the `.bss` section.

- If the *Small Data Area* (SDA) optimization is enabled (see “Special Data Area Optimizations” on page 137), the compiler re-allocates some data from the primary sections to the equivalent SDA sections:
 - `.sdata`
 - `.sbss`
 - `.sdata2`
 - `SMALLCOMMON`
- If the *Zero Data Area* (ZDA) optimization is enabled (see “Special Data Area Optimizations” on page 137), the compiler re-allocates some data items to the equivalent ZDA sections:
 - `.PPC.EMB.sdata0`
 - `.PPC.EMB.sbss0`
 - `ZEROCOMMON`



Note The placement of data depends on compile-time options.

Custom Program Sections

In addition to the default program sections, you can define your own custom program sections. Custom program sections allow you to group certain variables or functions in specific memory blocks on your target, using the following technique:

1. In your source code, assign text or data into a custom program section.
2. In your linker directives file, assign the custom section to a memory block on your target. If you want to initialize the data in a section to zero (as with a `.bss` section), use the `CLEAR` section attribute. For more information, see “Defining a Section Map with the `SECTIONS` Directive” on page 442.

For example, you might assign some variables that are accessed very often to a custom program section, and in your linker directives file, assign that section to a small block of very fast RAM. The following section explains how to assign text and data into a custom program section.

Assigning Data to Custom Program Sections in C

To assign data to a custom program section in C code, use the `#pragma ghs section` directive. This directive takes variables and functions that the linker would otherwise assign to the specified default section (such as `.data` or `.bss`) and assigns them to a custom section instead. In other words, it maps a type of data to a custom section. Each time you use the directive, it leaves mappings from earlier occurrences in place, except for those that it explicitly overrides. This directive must be placed before the definitions of the objects you want to assign to the custom section; placing it before a declaration has no effect. It should not be used inside of any function.

The syntax for the `#pragma ghs section` directive is:

```
#pragma ghs section [secttype="sectname"]
```

- *secttype* — A text string that represents the default section. Valid values for *secttype* are listed in the following table.
- *sectname* — The name of the custom section to which you want to re-map text or data, including the initial period character (.). If you specify the word `default` without quotes, it removes the mapping and restores the assignment rule to its initial state.
- If you do not specify *secttype*="sectname", all mappings are removed and section assignment rules are restored to their initial state.

secttype Value	Default Program Section
bss	.bss
data	.data
text	.text
rodata	.rodata
sbss	.sbss
sdata	.sdata
rodata2	.sdata2
zbss	.PPC.EMB.sbss0
zdata	.PPC.EMB.sdata0



Note For information about assigning objects to special data areas such as SDA, see “Special Data Area Optimizations” on page 137.

To change the compiler’s default section mappings, see “**Rename Section**” on page 160.

Example 2. Assigning Variables to Different Sections

This example assigns three different variables to three different sections:

```
/* Assign x1 to section .data1 */
#pragma ghs section data=".data1"
int x1 = 1;

/* Assign x2 to section .data2 */
#pragma ghs section data=".data2"
int x2 = 2;

/* Assign x3 to section .data3 */
#pragma ghs section data=".data3"
int x3 = 3;

/* Now go back to default rules */
#pragma ghs section data=default
```

Assigning Data to Custom Program Sections in Assembly

To assign data to a custom program section in assembly, use the `.section` directive. When defining a custom section, you must also specify the appropriate attributes:

- `a` — allocated to memory
- `w` — writable

For example, to define a section `.mydata` that will be laid out in RAM and assign the variable `foo` to that section, use the following assembly code:

```
.section ".mydata", "aw"
foo:
```

For a complete list of attributes for the `.section` directive, see “Section Control Directives” on page 421.

Reserved Section Names

The Green Hills run-time environment uses reserved program sections that are defined in default linker directives files. The run-time object files **libsys.a**, **libstartup.a**, and **crt0.o** use them to set up the run-time environment. If you are creating custom sections, do not duplicate the names of these reserved sections. For more information about them, see “Customizing the Run-Time Environment Program Sections” on page 459.

Assigning Program Sections to ROM and RAM

In embedded programming, it is generally the case that all program sections must be initially located in ROM to prevent their loss upon reset, and that any program sections containing data that may change at run-time must be copied into RAM at startup. You must decide how much of your program code will be copied into RAM at startup. Two factors that influence this decision are the amount of available RAM and the emphasis on speed of execution. In general:

- Programs that copy all of their sections from ROM to RAM at startup execute more quickly, but require more RAM.
- Programs that copy as few sections as possible from ROM to RAM at startup require less RAM, but execute more slowly.

Green Hills provides default linker directives files to manage the location of program sections. To specify one of them when creating a project with the **Project Wizard** (see “Creating A New Project” on page 22), set the **Program Layout** option to one of the following settings:

- **Link to ROM and execute out of RAM** — Uses the **standalone_romcopy.ld** linker directives file, which maximizes execution speed by copying all of the program sections from ROM to RAM at startup. If you are using the driver, specify the **-layout=romcopy** option to use this file by default.
- **Link to and execute out of ROM** — Uses the **standalone_romrun.ld** linker directives file, which minimizes RAM usage by copying as few program sections as possible from ROM to RAM at startup. If you are using the driver, specify the **-layout=romrun** option to use this file by default.
- **Link to and execute out of RAM** — [default] Uses the **standalone_ram.ld** linker directives file, which loads the program directly into RAM. Use this layout in the early stages of debugging, when you want to have the MULTI Debugger download your program to RAM and run it. If you are using the driver, specify the **-layout=ram** option to use this file by default.

Whichever layout you choose when creating your project, the **Project Wizard** puts all three linker directives files in the Target Resources project. To change your program layout, right-click your program’s project and select **Configure**.

These linker directives files provide a basic section map. If you want to use custom sections and control precisely where text and data are located, edit your linker directives file using the information in “Configuring the Linker with Linker Directives Files” on page 437. For further discussion about optimizing

your program by editing your linker directives file, see “Modifying your Section Map for Speed or Size” on page 458.

If you are using the driver with the **-layout=** option, you must also specify the **-bsp=** option (see “Specifying a PowerPC Target” on page 113). If no appropriate BSP is available, specify **-bsp=generic**. The **-layout=** option is ignored if you specify your own linker directives files.

The **-layout=** driver option is not supported with INTEGRITY. If you have an INTEGRITY project, set the program layout in the Project Manager or specify your linker directives files manually.

Storing Global Variables in Registers

Machine registers are primarily used by the compiler to store local variables or working values. This generally produces smaller and faster code than if most variables were on the stack. In certain circumstances, however, you can also allocate critical global data to registers (though this may decrease performance). Unless you are building INTEGRITY code, you can reserve up to 10 global registers (r14 to r23 in that order). To do so:



Specify the number of registers you want to reserve with the **Target→Text and Data Placement→Global Registers** (**-globalreg=*n***) option.



Note This option may not work correctly if you customize your startup code. If you are building INTEGRITY code, you cannot modify the default value of this option.

Setting this option allows the (otherwise illegal) C/C++ `register` keyword to place global variables in registers using the following syntax:

```
register int i; /* file scope implies global */
```



Note The syntax `register static int i` is still illegal, because it can cause a possible register numbering problem when compiling multiple files.

Global registers are initialized to zero by the startup module. The data type of the global variable can be any integral or pointer type where a local variable of that type would be eligible for allocation to a single register. It must not be explicitly initialized, and its address should never be taken. The program must be compiled with the Green Hills startup code.

Global registers are permanent registers, so functions compiled with this option can call (but not be called by) functions compiled without it, including third-party object files and libraries. A function may not access variables allocated to global registers unless all functions in the call chain before it were compiled with the global registers reserved. The easiest way to guarantee this is to compile all files with this option. However, since the Green Hills libraries are not compiled with this option, it is not safe to use variables allocated to global registers in library callback routines (such as the user-supplied comparison function called by the **stdlib.h** functions `bsearch()` and `qsort()`). It is

also generally not safe to use variables allocated to global registers in interrupt functions, as such interrupts may be triggered asynchronously.

If you want to share the same global registers across compilations, consider placing them into a single `#include` file, which is always included first in all compilations to ensure consistency.

Near and Far Function Calls

A function call is normally performed with an instruction that contains a 24-bit value that is shifted left by 2 and is added to the address of the current instruction to yield the address of the called function. If the offset to the called function is greater than 26 bits, a direct call instruction will be out of range.

To avoid this problem, the compiler provides support for *far function calls*. The compiler generates a series of instructions to place the address of the called function into a register before using an instruction to call the function. This method allows you to reach a function anywhere in the address space of the processor, but it requires additional code size and processor time.

Call Patching

By default, the linker handles far function calls by using a special feature called *call patching*. Call patching causes the linker to automatically detect function calls that are out-of-range and insert code between modules to resolve the far function call. The linker does this by creating a small patch of code, called a call patch, to perform a jump to the out-of-range function. It then changes the destination of the original function call to point to the call patch. This method impacts code size and run time less than using far function calls because it affects only those calls that are out-of-range. The one limitation of call patching is that the module containing the offending function call must be 64 MB or smaller; otherwise, the linker will not be able to place the call patch close enough to the function call.

Sometimes, you may want to rearrange code manually instead of using call patching. To disable call patching:



Set the **Target**→**Text and Data Placement**→**Linker-Based Far Call Patching** option to **Off** (`-nofarcallpatch`).

Far Function Call Build Option

To enable far function calls for every function:



Set the **Target→Text and Data Placement→Far Calls** option to **Make All Calls Far Calls (-farcalls)**.

This option instructs the compiler to generate a far function call for every call (with no need for modifications to the source code), unless the option is overridden by one of the other methods specified below. This method should only be used when far call patching is not an effective solution.

Function Call Pragma Directives

The following `#pragma` directives govern the declaration of functions. When you use one of these directives, all function declarations after it will be modified until another of these `#pragma` directives occurs.

```
#pragma ghs near
```

```
#pragma ghs callmode=near
```

Any function declared after this directive will be called using a near function call.

```
#pragma ghs far
```

```
#pragma ghs callmode=far
```

Any function declared after this directive will be called using a far function call.

```
#pragma ghs callmode=default
```

Any function declared after this directive will be called as directed by the **Target→Text and Data Placement→Far Calls** builder option (see “Text and Data Placement” on page 157).

Note that in certain cases, such as when both the caller and callee are located in the same file, the compiler may be able to determine that a far function call is unnecessary, and so perform a near call regardless of the `#pragma` directive.

Function Declaration

The keywords `__nearcall` and `__farcall` may be placed in the declaration of a function to specify the type of function call to be used for that

function. These keywords override the setting implied by either the `#pragma` directive or the command line option. For example:

```
__nearcall int nearfunc();  
__farcall int farfunc();
```

The syntax requires that `__nearcall` and `__farcall` be placed before the function return type.

The `__nearcall` and `__farcall` keywords cannot be used for function pointers, but only for functions themselves. They cannot be used in `typedef` statements, casts, or in the type of a function parameter.

Special Data Area Optimizations

Special data area optimizations place certain data in a special memory block that can be accessed by using an offset from a base address. In most cases, data in this block can be accessed more efficiently than other data. This optimization does not change the basic function of your program.

The PowerPC architecture supports *Small Data Area* (SDA) optimization and a specialized version of SDA called *Zero Data Area* (ZDA) optimization. This section describes these optimizations and explains how to allocate data to special data areas.

Small Data Area (SDA) Optimization

Small Data Area (SDA) optimization uses two special data areas, RAM SDA and ROM SDA, each containing 64 KB of data.

In general, the combined size of data segments that are assigned to SDA cannot exceed 128 KB. By convention, the Green Hills compiler uses register `r13` as the base register for the RAM SDA, and register `r2` as the base register for the ROM SDA. Because offset-based addressing on the PowerPC processor is signed, an SDA base register points 32 KB past the start of the SDA to provide a full 64 KB of addressing.

In order to implement SDA optimization, the compiler, assembler, and linker define the following special program sections:

- `.sdata` — initialized read/write data
- `.sbss` — uninitialized read/write data
- `.sdata2` — constant data

Zero Data Area (ZDA) Optimization

Zero Data Area (ZDA) optimization is a special case of SDA that uses 16-bit signed offsets to access memory around zero.

Due to the limitations on the offset-based addressing used to access ZDA variables, the size of the ZDA is limited to a total of 64 KB.

To implement ZDA optimization, the compiler, assembler, and linker define the following special program sections:

- `.PPC.EMB.sdata0` — initialized data
- `.PPC.EMB.sbss0` — uninitialized data

Assigning Data to Special Data Areas Using a Threshold

You can assign a threshold value (as an integer in bytes) so that the compiler assigns variables of that size or smaller to a special data area.

To specify an SDA threshold value:



Set the **Target→Text and Data Placement→Special Data Area** option to **Small Data Area** (`-sda[=threshold_value]`)

To specify a ZDA threshold value:



Set the **Target→Text and Data Placement→Special Data Area** option to **Zero Data Area** (`-zda[=threshold_value]`)

Where *threshold_value* must be one of the following:

- `none` — does not assign any data to the special data area using a threshold. If specified with `-zda`, this setting disables the `#pragma ghs startzda` and `#pragma ghs endzda` directives. If specified with `-sda`, this setting does not disable any `#pragma` directives.
- `0` — does not assign any data to special data areas using thresholds, but does not disable any `#pragma` directives. Use this option when you want to assign variables to special data areas with `#pragma` directives only.
- `n` — assigns all variables up to `n` bytes in size to the special data area. `n` must be a positive integer.
- `all` — assigns all variables to the special data area. While there is no size limitation, this option may cause the special data area to overflow.

`-sda` and `-zda` are mutually exclusive options; if you enable both of them on the command line, the last option takes effect



Note `ev64` variables are not assigned to special data areas, even if they are below the threshold or you specify `all`.

Assigning Specific Data To Special Data Areas in C

When programming in a high-level language, you can assign individual variables to a special data area by enclosing them in a set of `#pragma` directives. These directives are:

- For SDA — `#pragma ghs startsda / #pragma ghs endsda`
- For ZDA — `#pragma ghs startzda / #pragma ghs endzda`

The `startsda` and `startzda` directives cause the compiler to allocate all subsequent variables to the special data area, even if they exceed a specified threshold value (see “Assigning Data to Special Data Areas Using a Threshold” on page 138). An `endsda` or `endzda` directive ends this mode.

If a variable is declared between these `#pragma` directives and then declared again outside them, the compiler assigns the variable to the special data area.

If a variable is declared `extern` between special data area `#pragma` directives and later in the same module is defined as a global variable without them, it still goes into the special data area.



Note If you use these directives and set the option **-zda=none**, or if you are using INTEGRITY, the compiler issues warning 1510. ZDA is not implemented in INTEGRITY by default. If you have modified your INTEGRITY configuration to work with ZDA, set the **Target→Text and Data Placement→Special Data Area** option to a value other than none to clear this message and assign data to ZDA.

If you want only those variables explicitly specified by these `#pragma` directives to be allocated to the special data area, set the threshold value to 0 (see “Assigning Data to Special Data Areas Using a Threshold” on page 138).

Excluding Specific Data From Special Data Areas in C

To exclude data from a special data even if it is smaller than or equal to the specified threshold value, enclose the variable declarations in a `#pragma ghs startdata / #pragma ghs enddata` block. Variables defined within this block are never assigned to a special data area.

For example, if you set the threshold value to 8:

```
int foo_a = 4;           /* goes into SDA */
#pragma ghs startdata
int foo_b = 4;           /* does not go into SDA */
#pragma ghs enddata
```



Note This `#pragma` directive is ignored if you are using Automatic Link-Time SDA (see “Assigning Data to Special Data Areas Automatically” on page 141).

Assigning Specific Data To Special Data Areas in Assembly

When programming in assembly language, assign data to a special data area using the `.section` assembler directive. You must assign the appropriate attributes to that section.

For example, to include the variable `sda_var` in the `.sdata` section, enter the following in your assembly file:

```
.section ".sdata", "aw"
sda_var:
```

To include the variable `zda_var` in the `.PPC.EMB.sdata0` section of ZDA, enter the following in your assembly file:

```
.section ".PPC.EMB.sdata0", "aw"  
zda_var:
```

ROM SDA sections only require the `a` attribute, because they are not writable.

For a complete list of attributes for the `.section` directive, see “Section Control Directives” on page 421.

Assigning Data to Special Data Areas Automatically

If you want the toolchain to determine which data to allocate to special data areas, enable *Automatic Link-Time SDA*. This feature tells the linker to examine the entire program, and decide which data to allocate to special data areas. This can be preferable to compile-time allocation, because the linker knows about all variables and the number and type of accesses to them, and can use the link map to determine how much program memory is available for special data areas (up to the limit imposed by the target-specific addressing modes). To enable Automatic Link-Time SDA:



Set the **Target**→**Text and Data Placement**→**Automatic Link-Time SDA** option to **On** (`-auto_sda`).

If you are using INTEGRITY, remove the `.datachecksum` section from your default section map (see “Defining a Section Map with the SECTIONS Directive” on page 442).

The advantages of this method over compile-time allocation are as follows:

- It can use both SDA and ZDA if the relevant sections are defined in your link map.
- It allocates data to special data areas not based on size, but by the frequency with which they are accessed. Because this data benefits more from placement in a special data area, it often produces greater savings in speed and space than the compile-time method.
- It can calculate exactly how many data items can be allocated to special data areas, so it will always make the maximum use of these sections, but will never cause linker errors due to over-allocation.

When using this option, enable it at every stage of the build process (including compilation), because it inhibits some incompatible compiler optimizations. If you are linking previously compiled object files, re-specify

your **Optimization**→**Optimization Strategy** to ensure appropriate use of the SDA and ZDA sections.

When using Automatic Link-Time SDA, the linker assumes that the `.sdabase`, `.sdata`, and `.sbss` sections are contiguous, and tries to re-allocate up to 64K of data in the `.sdata` and `.sbss` sections combined. If the section map defines sections between `.sdabase` and `.sdata` or between `.sdata` and `.sbss`, or if `.sdata` and `.sbss` are not allocated into the same memory region, Automatic Link-Time SDA does not place anything in the `.sbss` section. The same caveats apply to the corresponding ZDA sections. We recommend that you do not modify the parts of the default section map that define SDA and ZDA sections (see “Customizing the Run-Time Environment Program Sections” on page 459).



Note When using Automatic Link-Time SDA:

- To force specific data objects into special data areas, use the `#pragma` directives described in “Assigning Specific Data To Special Data Areas in C” on page 139.
- To exclude specific data objects from being placed into special data areas, allocate them to other sections (see “Custom Program Sections” on page 128). The compiler ignores the `#pragma ghs startdata` and `#pragma ghs enddata` directives.

Editing Linker Directives Files for Use with SDA

The linker directives (`.ld`) file for a program that uses the SDA optimization must list SDA-related sections in a specific order. In particular, custom sections must be listed after `.sdabase`. The Green Hills tools define the base register(s) to point 32 KB past the start of `.sdabase` and assume that this address marks the middle of the SDA. If your user-defined section begins before `.sdabase`, its offset from the pointer will be greater than 32 KB and, therefore, will be inaccessible.

RAM SDA sections must be listed in the following order:

- Empty section used by Green Hills debug servers (`.sdabase`)
- Default SDA section for initialized data (`.sdata`)
- Custom SDA sections for initialized data, if any
- Default SDA section for zero-initialized data (`.sbss`)

- Custom SDA sections for zero-initialized data, if any

ROM SDA sections must be listed in the following order:

- Default ROM SDA section (`.sdata2`)
- Custom ROM SDA sections, if they exist

Example 3. Linker Directives File for an SDA-Optimized Program

The following is an example of a linker directives file for an SDA-optimized program that has a user-defined SDA section, `.mysdata`.

```
.text 0x10000 :  
.syscall :  
.secinfo :  
.rodata :  
.ROM.data ROM(.data) :  
.ROM.sdata ROM(.sdata) :  
.ROM.mysdata ROM(.mysdata) :  
.sdabase align(8) :  
.sdata :  
.mysdata :  
.sbss :  
.data :  
.bss :  
.heap align(16) pad(0x100000) :  
.stack align(16) pad(0x80000) :
```

Editing Linker Directives Files for Use with ZDA

You must specify the placement of the ZDA in a linker directives (`.ld`) file. The linker directives file will vary depending on the availability of ROM and RAM near location zero. For example, if RAM is available from address `0x0` through address `0x1000`, the following entries might be included in the linker directives file:

```
.PPC.EMB.sdata0 0x0 :  
.PPC.EMB.sbss0 max_endaddress(0x1000)
```



Note The ZDA sections `.PPC.EMB.sdata0` and `.PPC.EMB.sbss0` must be contained within the range `0xffff8000` to `0x00007fff`.

Assigning Data to Custom Special Data Area Sections

To assign variables to a custom special data area section, use `#pragma ghs` section within a block of C code. The following example assigns SDA variables to both the default SDA section `.sdata` and a custom SDA section `.mysdata`:

```
#pragma ghs startsda
/* puts variables into default SDA section */
int e = 3;
double f = 15.2;
#pragma ghs section sdata=".mysdata"
/* puts variables into .mysdata user-defined SDA section */
int g = 10;
short h = 3;
#pragma ghs endsda
```

For more information about `#pragma ghs` section, see “Assigning Data to Custom Program Sections in C” on page 129.

Linking Special Data Area Modules

Program source modules must agree about which variables are assigned to special data areas. While using just a threshold value avoids this problem, you must pay special attention to it when using `#pragma` directives to assign specific data. You may want to put all special data area `#pragma` directives and variable declarations into header files that are included by modules that use those variables.

In most cases, you can safely combine modules built with the SDA optimization with non-SDA modules. Since the ZDA base register `r0` is always available for ZDA, modules built with the ZDA optimization can also combine safely with non-ZDA modules. However, problems may arise when one of these special data area modules uses a global variable that is defined in a module that does not use the same special data area. If the module attempts to use base offset addressing to reference the variable, a link-time error can occur because the variable is not actually in a special data area. To correct this problem, recompile the modules with consistent use of special data areas.

Green Hills libraries are built with the SDA base register reserved. This allows both SDA and non-SDA modules to use the same libraries.

Linker Errors Related to Special Data Areas

The amount of data assigned to a special data cannot exceed the size of that data area. If data overflows, the linker issues an error. For SDA, the message is:

```
[elxr] (error) small data area overflow: offset (signed) didn't fit in n bits
      while performing relocation in file module
      at location location, to reference symbol symbol
      relocation type: type_num
```

ZDA provides a similar error message.

If you want to eliminate these errors by increasing the size of the special data areas, see “Specifying Additional Registers for Special Data Areas” on page 145. Another possible cause of these link errors is inconsistent use of the `#pragma` directives between modules. See “Assigning Specific Data To Special Data Areas in C” on page 139 for information about these directives.

In certain cases, the assembler is able to determine that the offset does not fit into 16 bits, and it prints a value too large for word sized cast error. This error is likely an indicator that too much data has been allocated to the ZDA section. Use the methods described in the previous sections to ensure that the end of the ZDA section is no more than 32 KB from address zero.

Specifying Additional Registers for Special Data Areas

Normally, in accordance with the PowerPC EABI, the compiler reserves registers `r2` and `r13` as SDA base pointers. To specify additional registers (`r14` through `r23`) to be used as SDA and ZDA base pointers:



Enter the number of additional base registers in the **Advanced**→**Target Options**→**Additional SDA Base Registers** (**-additional_sda_reg=*n***) option, where *n* is an integer between 1 and 10.

Registers are reserved sequentially starting with `r14`. Each additional register permits addressing 65,536 more bytes of data in a particular SDA section. The registers are used as base pointers to data that does not fit in the first 65,536 bytes of the `.sdata/.sbss`, `.sdata2/.sbss2` and

`.PPC.EMB.sdata0/.PPC.EMB.sbss0` sections. For example, setting this option to the value 2 reserves registers `r14` and `r15` in addition to `r13`. This option affects the compiler and linker, and must be passed to both.



Note This option has the following limitations:

- Using this option to reserve additional SDA base pointers makes them unavailable for normal use by the compiler, and may increase code size and/or degrade code speed.
- This option cannot be used in combination with:
 - the **Target→Text and Data Placement→Global Registers** (`-globalreg=n`) option (see “Storing Global Variables in Registers” on page 134)
 - the **Target→Text and Data Placement→Automatic Link-Time SDA** (`-auto_sda`) option (see “Assigning Data to Special Data Areas Automatically” on page 141)
- All modules must be compiled with the same number of additional SDA base registers.
- The ZDA sections `.PPC.EMB.sdata0` and `.PPC.EMB.sbss0` may not start at an address outside the range `0xffff8000` to `0x00007fff`.
- Variables in SDA sections cannot be accessed in library call-back functions (such as the user-supplied comparison function called by the **stdlib.h** functions `bsearch()` and `qsort()`).

Customizing the Green Hills Run-Time Environment

You can customize the Green Hills run-time environment to implement or enhance the environment for a particular hardware system by:

- Modifying the source code of the run-time object modules **libsys.a**, **libstartup.a**, and **crt0.o**, and then rebuilding these object modules (see “Customizing the Run-Time Environment Libraries and Object Modules” on page 751). Each source file, including files with only assembly language, is fully commented. These source files are located in the **src/libsys** and **src/libstartup** directories of your installation.
- Modifying the linker directives file for special program sections that **libsys.a**, **libstartup.a**, and **crt0.o** use to set up the run-time environment (see “Customizing the Run-Time Environment Program Sections” on page 459).

Other Topics

Renaming the Output Executable

To change the name of the output file:



Enter the desired name in the **Project→Output Filename (-o name)** option.

Specifying an Alternate Program Start Address

The linker normally uses the address of the global symbol `_start` as the start address for the user program. To specify an alternate start address:



Specify a new symbol in the **Linker→Start Address Symbol (-e=symbol_name)** option.

For example, to use the symbol `newstart` as the program start address, enter the following from the command line:

```
ccppc -e=newstart file.s
```

Writing Interrupt Routines

In C and C++, if you want a function to handle hardware interrupts or exception conditions, declare that function as an interrupt routine. When you declare a function as an interrupt routine, the compiler generates code in the function prologue that saves all the caller-save and callee-save registers it requires. It also generates code in the function epilogue that restores those registers to the state they were in before the function was called.

These functions should be of `void` type and should take no arguments.

You can use either of the following methods to declare a function as an interrupt routine:

- Place `#pragma ghs interrupt` immediately before the function.
- Prepend the `__interrupt` keyword to the function definition.



Note The PowerPC interrupt calling sequence is not compatible with the normal function calling convention. Calling an interrupt routine directly from user code may result in a memory access violation or other undefined behavior.

Establishing Interrupt Vectors with the `intvect` Pragma Directive

To establish an interrupt vector which is needed to call an interrupt routine in C and C++, use:

```
#pragma intvect function integer_constant
```

The compiler arranges for a jump to *function* to be placed in memory at the address specified by *integer_constant* using a `.org` directive to the assembler. The named *function* must be a valid interrupt routine that returns `void`.

Thus, the following code fragment declares an interrupt routine, `foo`, with an interrupt vector at address `0x60`, established by `#pragma intvect`:

```
__interrupt void foo(void);

#pragma intvect foo 0x60

__interrupt void foo()
{
    ...
}
```



Note This feature is supported only when using a Green Hills assembler and is not available in binary code generation mode. You must ensure that the call destination is within range of the vector or unpredictable behavior may result.

Symbolic Memory-Mapped I/O

Embedded systems may have memory locations reserved for I/O, with the registers of a hardware device mapped in these locations. Several techniques access such memory-mapped I/O devices, but using specially named sections has some advantages.

For memory-mapped I/O, the variables that correspond to the memory locations should be kept in one or more sections that contain only memory-mapped locations. These sections should not be initialized during program startup.

This approach creates a C language structure definition that matches the allocation of the I/O registers in the memory space. Named structure fields are assigned to correspond to the I/O registers. Next, the `#pragma` directive is used to define a variable of the `struct` type within a named `.data` section that holds only this variable. A linker section map places this named section at the position in memory of the I/O device. The I/O registers are now structure fields of the variable in the named section.

Structures should be defined with the `volatile` qualifier, because the registers' values can change without being modified explicitly by the program.

With this approach, the variable definition can be placed in a source file of its own. This file is compiled to an object file that is shared among different projects that need to refer to this I/O device. Each project needs the object file, an entry in the section map to position the section, and a header file to define the `struct` type.

In addition, you can then display the named variable in a MULTI Debugger data explorer to provide ongoing displays of the state of the I/O registers, and edit the fields of the data explorer to cause writes to the I/O registers. This displays and manipulates memory-mapped I/O registers by MULTI view windows, similar to built-in registers.

For example, consider a simplified universal asynchronous receiver-transmitter ("UART") that has three 16-bit registers: an output register, an input register, and a control/status register, mapped at locations `0x1000`, `0x1002`, and `0x1004` respectively. With code containing the following lines:

```
#pragma ghs section bss=".uartsec"
struct {
    short Tx;      /* Transmit Register      */
    short Rx;      /* Receive Register      */
    short CSR;     /* Control/ Status Register */
} volatile uart;
#pragma ghs section bss=default
```

and a section map entry in the linker directives file such as:

```
SECTIONS {
    ...
    .uartsec 0x1000 :
    ...
}
```

Code can then refer to the registers as `uart.Tx`, `uart.Rx`, and `uart.CSR`.

Within MULTI, the command `view uart` views a window showing the contents of the UART registers. Because MULTI must perform read operations to display this data, this approach is not suitable for registers that are sensitive to read and write operations (that is, registers for which accessing a register triggers an activity or referencing a data location near the register causes problems).

Verifying Program Integrity

If you want to verify that sections in memory are identical to those created by the linker, set the **Linker→Link-Time Checking→Checksum** option to **On (-checksum)**. This option instructs the linker to calculate a Cyclic Redundancy Check (CRC) checksum for each section that contains text or data and store it as the last 4 bytes of the section. To exclude a section from checksum creation, use the `NOCHECKSUM` attribute when defining that section in the section map.

To verify a section upon initialization, scan the `__secinfo` table, calculate the same CRC on all but the last 4 bytes of the section, and compare the result to the stored CRC. A match indicates that the program has the same byte values in memory that it had when the linker created the executable file. For more information about performing checksums, see `install_dir/src/libstartup/cksum.c`. For more information about the `__secinfo` structure, see `install_dir/src/libsys/indsecinfo.h`.

Builder and Driver Options

Chapter 6

This Chapter Contains:

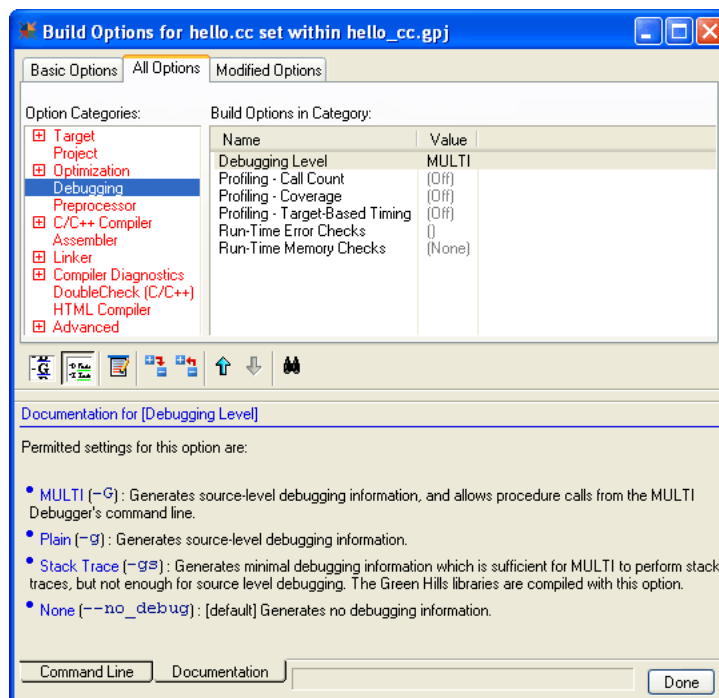
- Target Options
- Project Options
- Optimization Options
- Debugging Options
- Preprocessor Options
- C/C++ Compiler Options
- Assembler Options
- Linker Options
- Compiler Diagnostics Options
- DoubleCheck (C/C++) Options
- HTML Compiler Options
- Advanced Options

This chapter lists all of the Builder and driver options.

These options control various aspects of the compilation, assembly, and linking of your project. The following information is listed for each option:

- The name of the option as it appears in the **Build Options** window.
- A description of the option.
- Permitted option settings (if applicable).
- The equivalent driver option and description (if any).

To set options in the Project Manager, click **Edit** → **Set Build Options** to open the **Options** window (see “Setting Builder Options” on page 49).



Note The information in this chapter is available directly in the **Options** window on the lower **Documentation** tab (see “The Build Options Window Lower Tabs” on page 58).

To search for options in the **Options** window, click the **Search** button () and enter a text string to search for (see “Searching For Options” on page 59).

To pass options to the compiler driver, add them to your command line or makefile. For more information about using the driver, see Chapter 4, “The Compiler Driver”.

Target Options

This is a top-level option category.

These options allow you to control aspects of compilation relating to your target, such as available memory models and forms of floating-point support, and to control the instruction set to be used.

For additional, more specialized options, see “Target Options” on page 269.

Register Description File

Specifies a **.grd** file, which contains a description of the target board’s registers, and instructions on how MULTI may access them. The equivalent driver option is:

- **--register_definition_file=***file.grd*

Endianness

Controls the endianness of generated code. Permitted settings for this option are:

- **Big Endian (-bigendian)** — [default] The most significant byte of an integer appears at the lowest address.
- **Little Endian (-littleendian)** — The most significant byte of an integer appears at the highest address.

MULTI sets the default for this option based on the endianness of your board, which might not match the default setting for this option as indicated above.

Floating-Point

This option category is contained within **Target**.

These options control how floating-point operations are performed. The default setting depends on whether your target has a built-in *floating-point unit*, or FPU.

Floating-Point Coprocessor

Permitted settings for this option are:

- **No Floating-Point (-fnone)** — Disallows all floating-point operations (and directs the C and C++ compilers to give an error for any use of floating-point variables or constants), thus greatly reducing the size of such library functions as `printf` and `scanf`. Passes `-D__NoFloat__` to allow source code and header files to hide references to floating-point objects. If your code uses floating-point operations, then an error will be generated. This option implies **-nofloatio**. For more information, see “Predefined Macro Names” on page 664.

Because code related to floating-point operations may be linked in to your program even if it does not use floating-point variables, using this option may reduce the size of your program.

Because this setting requires support from the system libraries and header files, it is not supported for environments using embedded operating systems with their own C or C++ headers and libraries. In addition, certain target-specific header files may require the use of floating-point types.

- **Hardware Coprocessor (-fhard)** — Specifies *hardware floating-point* (HFP) mode, in which the compiler uses floating-point registers (where available) to hold floating point data, and floating point instructions to do floating point operations. This is the default for targets that have a built-in FPU, and is not available for other targets.
- **Software Emulation (-fsoft)** — Specifies *software floating-point* (SFP) mode, in which the compiler uses integer registers to hold floating-point data and generates library subroutine calls to emulate floating-point operations, regardless of the capabilities of the selected processor. This is the default for targets without an FPU. If the target has an FPU, this option selects a different set of floating-point libraries.

Treat Doubles as Singles

Controls the treatment of `double` types. Permitted settings for this option are:

- **On (-floatsingle)** — Treats `double` types as `float`, so that no 64-bit instructions are required. This option is only supported for processors with mixed floating-point support. To determine if your processor has this support, see “PowerPC Processor Variants” on page 114.



Note This mode does not fully comply with the C language specification. For example, the `double` data type does not have the required minimum precision or a precision greater than that of the `float` data type.

- **Off (-nofloatsingle)** — [default] Treats `double` types normally

Floating-Point Truncation

This option only affects the RS/6000 (**-cpu=rsc**), which does not have arithmetic instructions that produce a single precision result. Controls hardware floating-point precision. Permitted settings for this option are:

- **Precise (-fprecise)** — Truncates the results from single-precision arithmetic operations to their correct single-precision. This ensures that code which depends on the exact precision of single precision quantities will execute correctly.
- **Not Precise (-fnoprecise)** — [default] The results of single-precision arithmetic operations may be overly precise.

NaN Compares as Unordered

Generates extra instructions to ensure floating-point comparisons are correct for NaN operands. These extra instructions degrade floating-point comparison performance. This option only applies to PowerPC processors with the classic floating-point hardware unit (not the embedded floating-point hardware unit in e500v1 and e500v2 processors) and when hardware floating point is used **-fhard**. Permitted settings for this option are

- **On (-fNaN_cmp_unordered)** — Generate extra instructions to ensure floating-point comparisons are correct for NaN operands.
- **Off (-fno_NaN_cmp_unordered)** — [default] Do not generate extra instructions. Floating-point comparisons with NaN operands are only as accurate as the underlying hardware floating point comparison instruction guarantees.

Floating-Point Precise Signed Zero

The Green Hills tools generally conform to IEEE floating-point standards as much as possible. However, better code performance in both size and

speed can be obtained by allowing small deviations from the standard with respect to the treatment of signed zero quantities (-0.0 versus $+0.0$). When **-no_precise_signed_zero** is used, the compiler is permitted to replace sequences of operations that might have resulted in a negative zero (-0.0) outcome with one that may instead produce a positive zero result. Similarly, sequences that resulted in a positive zero result may be replaced with one resulting in a negative zero result. Consequently, because negative zero and positive zero are treated as equal quantities in every respect by the compiler, this should have no impact on the accuracy of user code when the distinction between signed zero quantities is of no importance. Permitted settings for this option are:

- **On (-precise_signed_zero)**
- **Off (-no_precise_signed_zero)** — [default]

Use Floating-Point in stdio Routines

Controls the use of floating-point in `stdio` operations. This option is deprecated and may be removed in future versions of **MULTI**. Permitted settings for this option are:

- **On (-floatio)** — [default] Use standard libraries.
- **Off (-nofloatio)** — Use **libnoflt.a**, a library containing special versions of `printf`, `scanf`, and related functions which, since they contain no floating-point operations, are therefore smaller. Floating-point formats (`%f`, `%g`, `%e`) are not supported. In environments where floating-point uses large library support routines, this option can save space for programs that use `printf`, but which do not require floating-point.

Text and Data Placement

This option category is contained within **Target**.

These options control the placement of text and data in memory.

Automatic Link-Time SDA

Controls the automatic allocation of data to special data areas (SDA and/or ZDA) at link-time. Permitted settings for this option are:

- **On (-auto_sda)**
- **Off (-no_auto_sda)** — [default]

For more information, see “Assigning Data to Special Data Areas Automatically” on page 141.

Special Data Area

Specifies that data up to a certain size should be placed in the *Small Data Area* or *Zero Data Area*. Permitted settings for this option are:

- **Small Data Area (-sda)** — Enables the *Small Data Area* optimization with a threshold of 8. For more information, see “Special Data Area Optimizations” on page 137.
- **Small Data Area with Threshold (-sda=size)** — Enables the *Small Data Area* optimization, where *size* specifies the threshold size for objects placed in the SDA.
- **Zero Data Area with Threshold (-zda=size)** — Enables the *Zero Data Area* optimization, where *size* specifies the threshold size for objects placed in the ZDA. For more information, see “Special Data Area Optimizations” on page 137.
- **No Special Data Area (-nothreshold)** — [default] Disables the special data area optimization.

Valid formats for the *size* parameter threshold include: 1024, 0x400, and all. If *size* is set to all, then no size restriction is set on objects placed into the special data area.

Far Calls

Controls whether the compiler will generate a far call for every call. Permitted settings for this option are:

- **Make All Calls Far Calls (-farcalls)** — Enables generation of a far function call for every call. Uses `mtctr/bctr1` instead of `b1` for all calls so that full 32-bit displacement is available. This allows for functions to be located at any distance from the caller.

- **Make No Far Calls (-nofarcalls)** — [default] Disables generation of far function calls. Large programs or programs with discontinuous text sections may not link if the range of the call instruction is exceeded.

Far calls are not available in combination with **Position Independent Code**.

For more information, see “Near and Far Function Calls” on page 135.

Linker-Based Far Call Patching

Controls a linker-based optimization which generates far calls only for calls which are out of range and would otherwise fail. This option will generally produce smaller code than generating far calls for every call (see “**Far Calls**” on page 158). Permitted settings for this option are:

- **Patch Far Calls When Necessary (-farcallpatch)**
- **Do Not Patch Far Calls (-nofarcallpatch)** — [default]

Placement of Zero-Initialized Data

Controls the allocation of statically-initialized variables and arrays that are explicitly initialized to zero. Allocating such objects to an uninitialized section will generally reduce the size of the executable ROM image. Permitted settings for this option are:

- **Place Zero-Initialized Data in BSS Sections (-discard_zero_initializers)** — Places statically-initialized variables that are explicitly initialized to zero as if they were defined but not initialized. For example, if `int x = 0;` would normally place `x` in the `.data` section, the compiler may place it in the `.bss` section instead, as if you had declared `x` with `int x;`. Regardless, `x` would still be initialized to zero.
- **Place Zero-Initialized Data in Data Sections (-no_discard_zero_initializers)** — [default] Places statically-initialized variables that are explicitly initialized to zero in the same way as those initialized to other values. If a global variable is not explicitly initialized in the source code, it may still be allocated to the appropriate `.bss` section.

Global Registers

Reserves up to 10 registers (r14 to r23 in that order) to hold global variables.

The equivalent driver option is:

- **-globalreg=*n***



Note If you are building INTEGRITY code, you cannot modify the default value of this option. If you attempt to do so, the compiler issues the following message:

Warning: The number of global registers on INTEGRITY is not configurable. option -globalreg=*n* ignored

For more information, see “Storing Global Variables in Registers” on page 134.

Stack Limit Checking

Controls stack limit checking, which is performed by making a call to a function `__stkchk()` in the prologue of each routine. Because these calls are added by the compiler, this option does not detect stack overflows that are caused by pre-compiled code, such as the Green Hills libraries. In stand-alone environments, `__stkchk` is in **libsys.a**. The code for it can be obtained in **src/libsys/ind_stackcheck.c** (see “Customizing the Run-Time Environment Libraries and Object Modules” on page 751). Permitted settings for this option are:

- **On (-stack_check)**
- **Off (-no_stack_check)** [default]

Rename Section

Assigns variables and functions to specific user-named sections. This option works in the same way as the `#pragma ghs section` directive but does not modify the source file (see “Custom Program Sections” on page 128). The equivalent driver option is:

- **-section *secttype=sectname***

secttype specifies which kind of code or data item is affected by the option and may be any of those listed in “Default Program Sections” on page 127 (with or without the leading period).

sectname is the user-defined section name, which starts with a period (.) by convention.

This option changes the default name of the specified section type throughout the file. Its effect is overridden by any `#pragma ghs` section in the source file with the same *secttype*, and its effect is restored by `#pragma ghs section secttype=default`.

Instruction Set

This option category is contained within **Target**.

These options control aspects of the instruction set.

Place 'EIEIO' Around Volatile Loads

Controls generation of `eieio` instructions after each load from a volatile variable. Permitted settings for this option are:

- **On** (`-eieio_loads`)
- **Off** (`-no_eieio_loads`) — [default]

The `eieio` instruction delays execution until all loads and stores complete. This option and **Place 'EIEIO' Around Volatile Stores** (see below) can be used to guarantee that loads and stores to volatile variables are executed in the correct order.

Place 'EIEIO' Around Volatile Stores

Controls generation of `eieio` instructions after each store to a volatile variable. Permitted settings for this option are:

- **On** (`-eieio_stores`)
- **Off** (`-no_eieio_stores`) — [default]

See also **Place 'EIEIO' Around Volatile Loads**, above.

VLE Code Generation and Libraries

Controls VLE code generation and linkage with VLE libraries. Permitted settings for this option are:

- **On (-vle)** — Enables VLE code generation and linkage with VLE libraries. This is the default for processors that default to the **ppc5514** library directory (see “PowerPC Processor Variants” on page 114).
- **Off (-novle)** — [default] Disables VLE code generation and linkage with VLE libraries. This option is the default for all other processors.

VLE code generation and linkage with VLE libraries is only allowed when a processor that supports the VLE extension is also selected. For a list of VLE-supported processors, see “PowerPC Processor Variants” on page 114.

Operating System

This option category is contained within **Target**.

These options are only available when your project specifies an embedded operating system. Additionally, some of these options are only relevant when you are creating ThreadX or INTEGRITY projects.

Shared Libraries

Controls whether the application may link in shared libraries. Permitted settings for this option are:

- **No Shared Libraries (-non_shared)**
- **Link With Shared Libraries (-call_shared)** — [default]

Additional System Library

Specifies an alternate system library, which is able to override parts of the default system library (**libsys**). **-syslib** can be used to allow an operating system to integrate with the C library without modifying the system library. The equivalent driver option is:

- **-syslib=library**

OS Directory

Specifies the root of the operating system distribution. If you plan to work on your project with both Windows and Linux/Solaris hosts, we recommend that you use forward slashes (/) in the path. The equivalent driver option is:

- **-os_dir** *directory*

Create RPL/RPX

This option is available for Cafe OS projects only.

Controls whether the driver creates an RPL file or an RPX file when linking a program. To create an RPL file or an RPX file, the driver invokes the **PrepRPL** and **MakeRPL** utilities from the Cafe SDK. Therefore, **-os_dir** must be set to the Cafe SDK root directory when enabling this option. Permitted settings for this option are:

- **Create RPL (-create_rpl)** — Invokes **PrepRPL** and **MakeRPL** to create an RPL file (not passing **-f** to **MakeRPL**).
- **Create RPX (-create_rpx)** — Invokes **PrepRPL** and **MakeRPL** to create an RPX file (passing **-f** to **MakeRPL**).
- **Off (-no_create_rpl)** — [default] Does not create an RPL file or an RPX file.

See the Cafe SDK documentation for further details about RPL files, RPX files, **PrepRPL**, and **MakeRPL**.

Use Source Root for RPX/RPL

This option is available for Cafe OS projects only.

Controls whether the driver passes the Source Root (see **Project→Source Root (-dbg_source_root)**) to **MakeRPL** when creating an RPL file or an RPX file. This requires Cafe SDK 2.11.02 or later. Providing the Source Root to **MakeRPL** will enable Cafe OS to provide relative paths to MULTI at run-time, which will help debug programs that are moved to a different machine or disk location after they are built. Permitted settings for this option are:

- **Use Source Root when creating RPX/RPL (-rpl_use_dbg_source_root)** — Specifies that the Source Root will be passed to **MakeRPL**.

- **Do not use Source Root when creating RPX/RPL**
(**-no_rpl_use_dbg_source_root**) — [default] Specifies that the Source Root will not be passed to **MakeRPL**.

This option has no effect if you are not creating an RPL file or an RPX file. See the Cafe SDK documentation for further details about RPL files, RPX files, **PrepRPL**, and **MakeRPL**.

Libraries and PrepRPL

This option is available for Cafe OS projects only.

Controls whether the driver passes implicitly included libraries to **PrepRPL** when creating an RPL file or an RPX file. Permitted settings for this option are:

- **Process all libraries (-preprpl_all_libs)** — Specifies that all libraries included in linking will also be passed to **PrepRPL**, even implicitly included libraries such as the C libraries.
- **Only process explicit libraries (-no_preprpl_all_libs)** — [default] Specifies that libraries will not be passed to **PrepRPL** unless they are explicitly included in the project (or, equivalently, if they are included by passing **-l** to the driver or by passing the library directly to the driver).

This option has no effect if you are not creating an RPL file or an RPX file. See the Cafe SDK documentation for further details about RPL files, RPX files, **PrepRPL**, and **MakeRPL**.

PrepRPL Export Object File

This option is available for Cafe OS projects only.

Specifies the name of the output **.o** file for **PrepRPL** when creating an RPL file or an RPX file. Regardless of whether this option is specified, the driver will pass **-o** to **PrepRPL** and link the resulting **.o** into the program. If this option is not specified, a temporary file will be used. The equivalent driver option is:

- **-preprpl_output=filename**

This option has no effect if you are not creating an RPL file or an RPX file. See the Cafe SDK documentation for further details about RPL files, RPX files, **PrepRPL**, and **MakeRPL**.

Additional PrepRPL Arguments

This option is available for Cafe OS projects only.

Specifies additional command line arguments for **PrepRPL** when creating an RPL file or an RPX file. The equivalent driver option is:

- **-preprpl.args=argument**

You can specify multiple arguments. Each item in the list in the **Build Options** window is exactly one command line argument (and is quoted by the Builder if required). Therefore, to pass arguments separately, it is necessary to enter them separately in the list. When using the driver, each argument must be specified with its own **-preprpl.args** option, for example:

```
-preprpl.args=-x -preprpl.args=filename
```

This option has no effect if you are not creating an RPL file or an RPX file. See the Cafe SDK documentation for further details about RPL files, RPX files, **PrepRPL**, and **MakeRPL**.

Additional MakeRPL Arguments

This option is available for Cafe OS projects only.

Specifies additional command line arguments for **MakeRPL** when creating an RPL file or an RPX file. The equivalent driver option is:

- **-makerpl.args=argument**

You can specify multiple arguments. Each item in the list in the **Build Options** window is exactly one command line argument (and is quoted by the Builder if required). Therefore, to pass arguments separately, it is necessary to enter them separately in the list. When using the driver, each argument must be specified with its own **-makerpl.args** option, for example:

```
-makerpl.args=-t -makerpl.args=BUILD_TYPE=NDEBUG
```

This option has no effect if you are not creating an RPL file or an RPX file. See the Cafe SDK documentation for further details about RPL files, RPX files, **PrepRPL**, and **MakeRPL**.

Event Logging

This option is available for ThreadX projects only.

Controls *event logging*, which collects event data for viewing and analysis in the MULTI EventAnalyzer (see “Viewing Trace Events in the EventAnalyzer” in Chapter 17, “Analyzing Trace Data with the TimeMachine Tool Suite” in the *MULTI: Debugging* book). Permitted settings for this option are:

- **On (-event_logging)** — Enables event logging by defining the `TX_EVENT_LOGGING` preprocessor symbol and linking with the **tx.e.a** ThreadX library.
- **Off (-no_event_logging)** — [default] Links with the standard **tx.a** ThreadX library.

Kernel Project

This option is available for INTEGRITY projects only.

Controls whether the project is to be linked as a kernel or as a standard application. Permitted settings for this option are:

- **On (Link with the default kernel) (-kernel)** — Specifies linking with the default INTEGRITY kernel library. This kernel library is fully featured and provides full debug support.
- **Off (-no_kernel)** — Specifies that no INTEGRITY kernel library be linked in.
- **Link with kernel optimized for production use (-kernel=kernel_opt)** — Specifies linking with a production-optimized INTEGRITY kernel library that does not include debugging features.
- **Link with kernel with checked debug output enabled (-kernel=kernel_checked)** — Specifies linking with an INTEGRITY kernel library that is fully featured, and provides extra debug output and internal error checking for debug purposes.
- **Link with velOSity kernel (-kernel=kernel_vel)**
- **Link with velOSity kernel with checked debug output enabled (-kernel=kernel_vel_checked)**
- **Link with velOSity kernel optimized for production use (-kernel=kernel_vel_opt)**

Include libbsp.a

This option is deprecated and provided for backwards compatibility only.

Controls whether your INTEGRITY kernel application will link against **libbsp.a**. Permitted settings for this option are:

- **On (-libbsp)** — [default]
- **Off (-no_libbsp)**

This option has no effect if you are not building an INTEGRITY kernel project.

Relocatable Module

This option is deprecated in INTEGRITY 5, and does not exist in INTEGRITY 10.

Creates a relocatable module, `mod`, that can be dynamically loaded to an already running `velOSity` system. Permitted settings for this option are:

- **On (-irel)**
- **Off (-no_irel)**

INTEGRITY Relocatable Program

Retains section-relative relocation information in the output file but removes relocations to weak undefined symbols. The resulting file is suitable for execution on INTEGRITY. Some of the final link steps, including but not limited to C++ constructors and special symbols, are not guaranteed to have relocations, and thus might not be valid if the output file is loaded at a different address. This option passes **-a** to the linker. Permitted settings for this option are:

- **On (-irelprog)**
- **Off (-no_irelprog)**

POSIX DLL

Generates an ELF executable format that is appropriate for INTEGRITY POSIX DLLs. Permitted settings for this option are:

- **On** (`--posix_dll`)
- **Off** (`--no_posix_dll`)

Relocatable Module Base Image

This option is deprecated in INTEGRITY 5, and does not exist in INTEGRITY 10.

Specifies an alternate kernel image if the default kernel is not used (see “Building Relocatable Modules” in the *velOSity User’s Guide*). The equivalent driver option is:

- **-dyload**

Full POSIX-2001 Compilation Support

Enables the C99 keyword support for POSIX. Permitted settings for this option are:

- **On** (`--integrity_posix`)
- **Off** (`--no_integrity_posix`)

INTEGRITY Posix Mode

Simplifies what libraries to link in, especially when dealing with shared libraries and POSIX. Permitted settings for this option are:

- **No POSIX Libraries** (`:integrity_posix_mode=noposix`)
- **Single Address Space POSIX Libraries** (`:integrity_posix_mode=single`)
- **Full POSIX System Libraries** (`:integrity_posix_mode=system`)

For more information, “Introduction to POSIX and INTEGRITY” in the *INTEGRITY Libraries and Utilities User’s Guide*.

Dynamic Download Project

This option is available for INTEGRITY projects only.

Controls whether the project will be linked to be suitable for downloading to a running kernel. Permitted settings for this option are:

- **On (-dynamic)** — Is ignored if an **.int** file is specified.
- **Off (-no_dynamic)** — [default]

Display BootTable

This option is available for INTEGRITY projects only.

Controls the build-time display of the BootTable that the kernel processes at run time. Permitted settings for this option are:

- **On (-display_boot)**
- **Off (-no_display_boot)** — [default]

Delete Unused Kernel Calls

This option is available for INTEGRITY projects only.

Controls the deletion of unused kernel calls from the final image. Permitted settings for this option are:

- **On (-generatemonolithdeletions)**

The deletion is based on physical and virtual kernel calls referenced in the component `AddressSpaces`. This option may cause unpredictable errors if used in the compilation of a dynamic download project which could use a deleted kernel call.

Create Raw Binary Image

Controls the creation of a binary image in addition to the standard ELF image. This option applies only to projects that contain an Integrate (**.int**) file. If you want to generate a binary image for anything else, see “**Generate Additional Output**” on page 244. Permitted settings for this option are:

- **Generate Default Binary Image (-memory)** — Generates a binary memory image output file with the name of the ELF executable plus a **.bin** extension.

- **Generate User-Specified Binary Image (-memory=filename)** — Creates a binary image with the specified *filename*.

Additional Intex Options

This option is available for INTEGRITY projects only.

Specifies additional **Intex** options. The equivalent **intex** option is:

- **-intexoption** *option*

Generated Header File Directory

This option is available for INTEGRITY projects only.

Sets the base directory to use for header file generation. Default is the directory where the **.bsp** file resides. The equivalent **intex** option is:

- **-headerfiledir** *dir*

Generated Header File Name

This option is for INTEGRITY projects only.

Sets the name for the default header file to use for header file generation. Default is *basename_of_.int_file_integrate.h*. The equivalent **intex** option is:

- **-headerfilename**

For more information about header file generation, see the "Header File Generation" section in the *Integrate User's Guide*.

Include File Search Directory

This option is available for INTEGRITY projects only.

Adds the specified directory to the search path for `%include` files. The equivalent **intex** option is:

- **-includefiledir**

INTEGRITY Application Stripping

This option is available for INTEGRITY projects only.

Controls *stripping* of the executable at the conclusion of linking. Permitted settings for this option are:

- **On** (**-strip_application**)
- **Off** (**-no_strip_application**) — [default]

Intex Variable Definition

This option is available for INTEGRITY projects only.

Overwrites an entry in a `Variables` section. *name* indicates the name of the *Variable* to be overwritten. *value* indicates the new value of the variable. The equivalent **intex** option is:

- **-intex_var** *name=value*

Quit Building if Intex Warnings are Generated

This option is available for INTEGRITY projects only.

Controls whether building will cease upon the generation of any warning. Permitted settings for this option are:

- **On** (**--quit_after_intex_warnings**)
- **Off** (**--no_quit_after_intex_warnings**)

SHA-1

This option is available for INTEGRITY projects only.

Prepares the final image for an SHA-1 consistency check upon boot. Use this option in conjunction with **libsha1.a** in the INTEGRITY kernel. Permitted settings for this option are:

- **On** (**-sha1**)

Project Options

This is a top-level option category.

These options allow you to control basic configuration settings for your project, such as source, output and include directories.

For additional, more specialized options, see “Project Options” on page 273.

Object File Output Directory

Puts object files into the specified *directory*, which is often a subdirectory of the current working directory. Along with the object files, the assembly listings, debugging information files, inliner files, and other intermediate files that have the same base name as the object file but a different suffix are also put into this directory. Note that the output of the linker and the output of the archiver are never put into the object file output directory. If this option is not specified, the object files will be put into the current working directory.

Multiple projects may not share an object directory.

The equivalent driver option is:

- **-object_dir**=*directory*

Source Root

Specifies a source root for use by the MULTI Debugger **sourceroot** command (see MULTI: Debugging Command Reference). The Debugger resolves paths relative to this root. The equivalent driver option is:

- **-dbg_source_root** *path*

Include Directories

Specifies a *directory* in which the builder should search for header files. The equivalent driver option is:

- **-Idirectory**

When the compiler processes source files that have `#include "header_file"` or `#include <header_file>` directives, it looks for *header_file* in any directories specified with this option. You can specify multiple directories, and the compiler will search them in the order in which they are specified. If the source file's `#include` directive declares a full path to the header file, then the compiler ignores any directories specified with this option.

For a detailed discussion of how the compiler searches for included files, see “Instructing the Compiler to Search for Your Headers” on page 122.

Do not use this option to specify the location of Green Hills header files. The driver selects these files automatically, based on other options.

Library Directories

Specifies a *directory* in which the builder searches for libraries specified by the `-I` option. The equivalent driver option is:

- `-Ldirectory`

All `-L` options on the command line are processed before the linker searches for the libraries specified by `-I`. If you use this option multiple times, the builder searches directories in the order in which they appear on the command line.

If you are not using the default startup code, note that this option changes only the search path for libraries. This means that the linker always takes **crt0.o** from the default location, even if it finds **libstartup.a** and **libsys.a** in another specified location. For more information about how to configure your project to use custom startup code, see “Settings for Stand-Alone Programs” on page 31.

Do not use this option to specify the location of Green Hills libraries. The driver selects these libraries automatically, based on other options.

Libraries

Specifies a *library* for the builder to link against. The equivalent driver option is:

- `-lname`

If *name* does not have an extension or directory path, it is assumed to be an abbreviation for **libname.a**. If *name* does not have a directory path, the driver

and linker searches for **libname.a** in the directories specified by the **-Ldir** option in the order that the **-L** options were specified, followed by a list of default directories.

If *name* has a directory path, *name* is assumed to be the complete filename of a library or object file that is passed directly to the linker without the **-l** option.

You can specify multiple libraries, and they will be linked against in the order specified. When using the driver, each library must be specified with its own **-l** option, for example:

```
-lfoo -lbar -lbaz
```

To prevent multiply defined symbols, it is recommended that you list your source files on the command line first, and then any **-llibrary** libraries.

Do not use this option to specify the location of Green Hills libraries. The driver selects these libraries automatically, based on other options.

Output Filename

Names the generated output file. The output file type (for example, a library or an assembly file) depends on the other options that have been specified. The builder enforces specific suffixes for some types of output files. This option has no effect when set on a Top Project. The equivalent driver option is:

- **-o filename**

Unless there is a single input file on the command line, you cannot use **-o** with **-c**, **-E**, **-P**, **-Q**, or **-S**.

Source Directories Relative to This File

Specifies a *directory* in which the Builder should search for source files. The syntax for this option is:

- **:sourceDir=directory**

The Builder searches for source files in the same directory as the project file that specifies them before searching directories specified by **:sourceDir**.

This option controls behavior in the Builder. There is no equivalent driver option.

Intermediate Output Directory Relative to Top-Level Project

Specifies the path where object files (and, in addition, any custom output file types) are written to, relative to the location of your Top Project (usually **default.gpj**). The syntax for this option is:

- **:outputDir**=*directory*

This option is most useful for custom tools that do not allow you to specify a directory for intermediate output. For projects that use only Green Hills Tools, use **-object_dir** instead. **-object_dir** overrides this option.

This option controls behavior in the Builder. There is no equivalent driver option.

Optimization Options

This is a top-level option category.

This section covers the wide range of optimizations Green Hills provides to help you produce smaller and/or faster executables.

Optimization Strategy

Specifies the high-level optimization strategy that MULTI uses when compiling your project. Permitted settings for this option are:

- **Maximum Debugging And No Inlining (-Omaxdebug)** — Disables inlining and all optimizations.
 - *Debugging ability* — excellent
 - *Code size and speed* — not optimized
 - *Compilation speed* — moderately fast
 - *Intended use* — development for code that has a lot of inline functions
- **Maximum Debugging And Limited Optimizations (-Omoredebug)** — [default] Enables optimizations that do not affect debugging.
 - *Debugging ability* — excellent (except for inline functions)
 - *Code size and speed* — slightly optimized
 - *Compilation speed* — moderately fast
 - *Intended use* — development; production for projects that are not size restricted and that use the same builds as development
- **Optimize for Debuggability (-Odebug)** — Enables optimizations that do not affect debugging.
 - *Debugging ability* — good (debug information is unavailable in rare cases)
 - *Code size and speed* — moderately optimized
 - *Compilation speed* — fast
 - *Intended use* — development; production for projects that are not size restricted and that use the same builds as development
- **Optimize for General Use (-Ogeneral or -O)** — Enables optimizations that improve both size and performance.

- *Debugging ability* — moderate (debug information is occasionally inaccurate)
- *Code size and speed* — optimal for general performance
- *Compilation speed* — moderate
- *Intended use* — production, when speed and size are equally important
- **Optimize for Size (-Osize)** — Enables optimizations that improve both size and performance, and additional optimizations that improve code size at the expense of performance. When using this strategy, apply it as globally as possible. The **-Ospace** option is equivalent to **-Osize**.
 - *Debugging ability* — moderate (debug information is occasionally inaccurate)
 - *Code size and speed* — optimal size, moderately fast
 - *Compilation speed* — moderate
 - *Intended use* — production, for size-restricted projects
- **Optimize for Speed (-Ospeed)** — Enables optimizations that improve both size and performance, and additional optimizations that improve performance at the expense of size. We recommend that you apply this strategy only for files and functions that use the most program execution time. You can then set a different optimization strategy globally (on your project file) to get multiple benefits.
 - *Debugging ability* — moderate (debug information is occasionally inaccurate)
 - *Code size and speed* — moderately small, optimal speed
 - *Compilation speed* — moderate
 - *Intended use* — production, when speed is more important than size
- **No Optimizations (-Onone)** — Disables all optimizations and provides the most straightforward code generation. Inlining of explicitly marked functions is performed at this level.
 - *Debugging ability* — good
 - *Code size and speed* — not optimized
 - *Compilation speed* — fastest
 - *Intended use* — when compilation speed is your highest priority, or when instructed by Green Hills support

When deciding how to optimize your program, always begin by selecting the optimization strategy that is the best fit for your project. If you must fine-tune a strategy to your specifications, see Chapter 7, “Optimizing Your Programs” and “Optimization Options” on page 287.

Intermodule Inlining

Enables *two-pass inlining*. If no optimization strategy is selected, this option enables **-Ospeed**. Permitted settings for this option are:

- **On (-OI)**
- **Off (-Onoinline)** — [default]

For more information, see “Inlining Optimizations” on page 317.

Automatic Vector Optimization

Controls Vector Optimizations. If no optimization strategy is selected, this option also enables **-Ospeed**. Permitted settings for this option are:

- **On (-OV)**
- **Off (-Onovector)** — [default]

For more information, see “SIMD Vectorization” on page 343.

Linker Optimizations

Controls the linker optimizations listed in “Linker Optimizations” on page 250, together with **Target→Text and Data Placement→Automatic Link-Time SDA** (see “Text and Data Placement” on page 157). Permitted settings for this option are:

- **On (-Olink)** — The linker optimizations that are enabled depend upon whether you are optimizing for speed or size. Many of these optimizations make complex changes to your code, and might slow down the link stage, be harder to debug, or have other drawbacks. To disable individual linker optimizations, set them to **Off** in combination with this option.

-Olink implicitly enables **-delete** and **-uvfd**. When building shared objects, this could result in unresolved symbols. For more information about linker-specific options, see “Linker-specific Options” on page 434.

For more information, see “Deleting Unused Functions” on page 467.

- **Off (-Onolink)** — [default]

Interprocedural Optimizations

Performs optimizations based on all available functions, unlike most other optimizations that only consider one function at a time. Permitted settings for this option are:

- **Wholeprogram (-Owholeprogram)** — Enabling wholeprogram optimizations allows program control and data flow to be analyzed at a high level. Speed and size optimizations such as one call-site inlining, interprocedural constant propagation and dead code elimination, and interprocedural alias analysis are performed. This option can improve program speed and size, often simultaneously.
- **Interprocedural (-Ointerproc)** — Enabling interprocedural optimizations allows optimizations based on knowledge of functions being called, such as interprocedural alias analysis. Unlike **-Owholeprogram**, **-Ointerproc** does not require that the entire program is available during compilation. However, a strict subset of the optimizations from **-Owholeprogram** are applied with **-Ointerproc**.
- **Analysis without optimizations (-Oip_analysis_only)** — Performs interprocedural analysis but does not apply any optimizations.
- **Off (-Onoipa)** — Disables all interprocedural optimizations.

For more information about using interprocedural optimizations, see “Interprocedural Optimizations” on page 330.

Optimization Scope

This option category is contained within **Optimization**.

These options provide general control over the major optimizations.

Pipeline & Peephole Scope

Restricts the scope of optimizations to ensure the retention of debugging information. Permitted settings for this option are:

- **Restrict peephole optimization scope to increase debuggability**
(-Olimit=peephole)
- **Restrict pipeline optimization scope to increase debuggability**
(-Olimit=pipeline)

Inline Larger Functions

Controls whether the compiler will consider larger functions when **-OI** is enabled. If no optimization strategy is selected, this option enables **-Ospeed**. Permitted settings for this option are:

- **On (-OB)**
- **Off (-Onobig)** — [default] You can use `-Onobig` in a MULTI Project (.gpj) file, but there is no equivalent driver option.

For more information, see “Varying Inlining Thresholds” on page 324.

Unroll Larger Loops

Controls the size of the loops that the compiler will consider for unrolling. Permitted settings for this option are:

- **On (-Ounrollbig)** — Consider larger loops for unrolling.
- **Off (-Onounrollbig)**

For more information, see “Loop Unrolling” on page 342.

Maximize Optimizations

Controls the aggressiveness with which the compiler will pursue optimizations, without regard for compile time. Permitted settings for this option are:

- **On (-Omax)**
- **Off (-Onomax)** — [default]

For example, to optimize most aggressively for size, without regard for compilation time, set **Optimization**→**Optimization Strategy** to **Optimize for Size (-Osize)** and set this option to **On**.

Symbols Referenced Externally

This option should only be used if you are optimizing with wholeprogram optimizations, and the entire program is not available during each compilation. This often occurs if you have assembly files or other code that is linked in. Because wholeprogram optimizations rely on the knowledge of all function and variable uses, if a function or variable is referenced externally (for example, from an assembly file) you must list that symbol with **-external**.

If you are using linker optimizations and want to prevent a function from being deleted, you may also need to pass the **-keep** linker option. For more information about linker-specific options, see “Linker-specific Options” on page 434.

The equivalent driver option is:

- **-external=function1 -external=function2 -external=variable1**

function1, *function2*, and *variable1* are functions and variables in the source code that can be referenced externally.

Files Listing External Symbols

Takes a file listing externally visible symbols rather than taking the symbol names explicitly. If you have many symbols that can be referenced externally, you might want to list them in a file and pass this file to the compiler rather than specifying each symbol as shown above.

The equivalent driver option is:

- **-external_file=file1 -external_file=file2**

file1 and *file2* are the complete paths of the files listing symbols that can be referenced externally.

These files can contain comma- or white-space separated symbol names listing those symbols that are externally visible. Additionally, these files can be

composed of global symbols defined or used in object files and libraries that are not specified during compilation, but are linked into the final executable. A list of these symbols can be generated by using the **gnm** utility (see “The gnm Utility Program” on page 531 for more information).

IP One-Site Inlining

Inlines functions that only have one call-site. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oiponesiteinlining**)
- **Off** (**-Onoiponesiteinlining**)

IP Delete Functions

Deletes unreachable functions. With or without this option enabled, functions may be removed if all of their uses are inlined. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipdeletefunctions**)
- **Off** (**-Onoipdeletefunctions**)

IP Delete Globals

Deletes constant-value global variables that are not indirectly referenced. This option is only meaningful if you pass the **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipdeleteglobals**)
- **Off** (**-Onoipdeleteglobals**)

IP Constant Globals

When a global variable is initialized to a constant value and never changed, and the variable is not indirectly referenced, the initial value is propagated into the

uses of the variable. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipconstglobals**)
- **Off** (**-Onoipconstglobals**)

IP Small Inlining

Functions are inlined when the inlined size might be less than the size of the call to the function. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipsmallinlining**)
- **Off** (**-Onoipsmallinlining**)

IP Constant Propagation

Conventional constant propagation is performed on the arguments of function calls. This option is only meaningful if you pass the **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipconstprop**)
- **Off** (**-Onoipconstprop**)

IP Remove Parameters

When certain parameters are not required, the parameters are removed, and function calls do not pass arguments for them. This option is only meaningful if you pass the **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipremoveparams**)
- **Off** (**-Onoipremoveparams**)

IP Limit Inlining

Limits the amount of inlining performed by interprocedural optimizations. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oiplimitinlining**)
- **Off** (**-Onoiplimitinlining**)

IP Constant Returns

When functions return constant values, those values can be propagated to the caller. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipconstantreturns**)
- **Off** (**-Onoipconstantreturns**)

IP Remove Returns

When the result of a function call is not required, the return statement is removed. This option is only meaningful if you pass the **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipremovereturns**)
- **Off** (**-Onoipremovereturns**)

IP Alias Reads

Determines what memory can be read by function calls. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipaliasreads**)
- **Off** (**-Onoipaliasreads**)

IP Alias Writes

Determines what memory can be written by function calls. This option is only meaningful if you pass the **-Ointerproc** or **-Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipaliaswrites**)

- **Off** (**-Onoipaliaswrites**)

IP Alias Lib Functions

Performs interprocedural alias analysis on some common library functions such as `printf`, `strlen`, and `fopen`. This option is only meaningful if you pass the **-Ointerproc** or **Owholeprogram** option. Permitted settings for this option are:

- **On** (**-Oipaliaslibfuncs**)
- **Off** (**-Onoipaliaslibfuncs**)

Individual Functions

This option category is contained within **Optimization**.

These options allow you to specify individual functions for optimization.

Optimize Specific Functions for Speed

Enables speed optimizations for specific functions. The equivalent driver option is:

- **--fast=***function1*[*function2*...]

Note that this option has no effect unless an overall **Optimization**→**Optimization Strategy** is specified.

Optimize Specific Functions for Size

Enables size optimizations for specific functions. The equivalent driver option is:

- **--small=***function1*[*function2*...]

Note that this option has no effect unless an overall **Optimization**→**Optimization Strategy** is specified.

Inline Specific Functions

Enables two-pass inlining for specific functions. The equivalent driver option is:

- **-OI=***function1[,function2...]*

For more information, see “Inlining Optimizations” on page 317.

Loop Optimize Specific Functions

Enables loop optimizations for individual *functions*. If no optimization strategy is selected, this option enables **-Ospeed**. The equivalent driver option is:

- **-OL=***function1[,function2...]*

For more information, see “Loop Optimizations (including SIMD Vectorization)” on page 339 and “SIMD Vectorization” on page 343.

Debugging Options

This is a top-level option category.

These options control the generation of debugging and profiling information. For more information, see “Generating Debugging Information” on page 115 and “Obtaining Profiling Information” on page 117.

For additional, more specialized options, see “Debugging Options” on page 292.

Debugging Level

Permitted settings for this option are:

- **MULTI (-G)** — Generates source-level debugging information, and allows procedure calls from the MULTI Debugger’s command line.
- **Plain (-g)** — Generates source-level debugging information.
- **Stack Trace (-gs)** — Generates minimal debugging information sufficient for MULTI to perform stack traces, but not sufficient for source-level debugging. The Green Hills libraries are compiled with this option. This option also causes **-gtws** to be enabled by default (normally **-nogtws** is enabled by default). You can override this with the **-nogtws** option. For more information, see “Debugging Options” on page 292.
- **None (--no_debug)** — [default] Generates no debugging information.

Profiling - Call Count

Controls *call count* profiling, which records how many times each function is called. Permitted settings for this option are:

- **On with Call Graph (-pg)** — Generates call count information that includes the name of each caller. This information can be used to generate a call graph.
- **On (-p)** — Generates basic call count information.
- **Off (-pnone)** — [default] Disables call count profiling.

Profiling - Coverage

Controls *coverage* profiling, which records whether your application contains lines of code that are not used. Permitted settings for this option are:

- **On** (-a)
- **Off** (--no_coverage_analysis) — [default]

Coverage profiling might not be reliable when used in conjunction with compiler optimizations, link-time optimizations, and options that cause code instrumentation (for example, run-time error checking).

Profiling - Target-Based Timing

Controls *timing* profiling, which records the time spent in each function. You should only use this option in stand-alone and u-velOSity projects. Permitted settings for this option are:

- **On** (-timer_profile)
- **Off** (-no_timer_profile) — [default]

Enabling target-based timing profiling requires that timer code be present and running on the target. For more information, see “Target-Based Timing Profiling” on page 120.

Run-Time Error Checks

Controls run-time error checking, which can be helpful during a debugging session, but which increases the size and reduces the speed of your application. Run-time error checks catch many occurrences of the common errors described below. The syntax for this option is:

-check=type [, type] ...

-check=type [-check=type]...

Enables run-time error checks for each specified *type*.

-check=all [, notype] ...

-check=all [-check=notype]...

Enables all run-time error checks. To enable all checks except those of a certain type, follow this option with `, notype` or `-check=notype`.

Run-time error checks are also enabled by MISRA 2004 rule 21.1 and MISRA 1998 rules 4 and 107. The various checks generate warnings or errors as specified below. Permitted values for *type* are:

- **All (-check=all)** — Enables all run-time error checks. This option does not affect run-time memory checks.
- **None (-check=none)** — [default] Disables all run-time error checks. This option does not affect run-time memory checks.
- **Assignment Bounds (-check=assignbound)** — Generates a warning if a value assigned to an integral or enumerated object is out-of-bounds for the object being assigned to.
- **Array Bounds (-check=bounds)** — Generates an error if an array is accessed with an invalid index. Only arrays whose bounds are known at compile time are checked; incomplete and variable-length arrays (VLAs) are not checked. The compiler checks each individual array index separately. For example:

```
int a[2][3];
a[0][3] = 0; // produces a run-time error
a[1][0] = 0; // equivalent behavior, no run-time error
```

This option does not output a run-time check for the following example, because the `arr` parameter's type decays into `int*` and could legitimately point to an array of any size:

```
int func(int arr[20], int i)
{
    return arr[i];
}
```

- **Case Label Bounds (-check=switch)** — Generates a warning if the `switch` expression does not match any of the `case` labels. This does not apply when a default `case` label is used.
- **Divide by Zero (-check=zerodivide)** — Generates an error indicating that a divide by zero occurred and then terminates the program. This option instruments both integer and floating-point divides.
- **Nil Pointer Dereference (-check=nilderef)** — Generates an error for dereferences of null pointers.

- **Write to Watchpoint (-check=watch)** — Instruments your code to watch writes to one arbitrary address in memory. Use this check when your target does not support hardware breakpoints. Specify the watchpoint address at run time with the **watchpoint** command. For more information about watchpoints, see Chapter 4, “Breakpoint Command Reference” in the *MULTI: Debugging Command Reference* book.

To enable or disable all the checks when using the Builder, use the **All** toggle switch at the bottom of the dialog box. When using the driver, pass the option **-check=all** or **-check=none**.



Note **-check=assignbound** and **-check=switch** produce run-time errors on code that is legal in C and C++, but may be considered to have bad style.

You can perform run-time error checking without the MULTI Debugger by rebuilding your application with run-time error checking enabled and running it normally. The program reports errors and warnings to `stderr`. After an error, the program calls `exit` with a value greater than 0 (or `HaltTask`, if it is an INTEGRITY program), terminating itself. After 101 warnings are produced, it prints a message indicating too many errors and calls `exit` with a value greater than 0 (or `HaltTask`), terminating itself.

You can also hook the run-time error checking reporting to automatically invoke your own handler for any failures that are detected. The following is an example of how to set up this feature:

1. Create a new project (see “Creating A New Project” on page 22).
2. Select your **Operating System, Processor Family, and Target Board**.
3. After creating a new project, click  to add an executable or examples to your Top Project.
4. From the **Select Item to Add** dialog box, select **Demo: Custom Run-Time Error Checking** and click **Finish**.

If you would like to select the directory and name of your new item, click **Next**

- a. Select the directory and name of your project and click **Next**.
- b. Select your program layout, and custom libraries, and click **Finish**.

5. Scroll down the project tree and open **user_defined_handler.c**. You will find sample code that will help you set up custom run-time error checking without the MULTI Debugger.

Run-Time Memory Checks

Controls run-time memory checking. Permitted settings for this option are:

- **General (Allocations Only) (-check=alloc)** — Selects a special version of memory allocation library routines, including `malloc` and `free`. These library routines detect a limited number of memory allocation problems, such as freeing an object that has not been allocated. This option only changes the way the program is linked and does not change the code produced by the compiler.
- **Intensive (-check=memory)** — Introduces additional code in the user application at each dereference of a pointer to detect a wide range of memory allocation problems, including those detected by **-check=nilderef**. This setting implies **-check=alloc** because the special library routines selected by that option are required. To perform full memory checking on a small part of an application, specify **-check=memory** when compiling the files or files of interest, and specify **-check=alloc** when linking the entire program.
- **None (-check=nomemory)** — [default] Disables memory checking.

For detailed information, see Chapter 14, “Viewing Memory Allocation Information” in the *MULTI: Debugging* book.

Preprocessor Options

This is a top-level option category.

These options control the preprocessor. For information about the Green Hills predefined macros, see “Predefined Macro Names” on page 664.

For additional, more specialized options, see “Preprocessor Options” on page 299.

Define Preprocessor Symbol

Defines a preprocessor *symbol*, and optionally sets it to a *value*. When setting this option in the **Edit List Option** window in the Project Manager, type *symbol=value* in the text field. The equivalent driver option is:

- **-Dsymbol[=value]**

This is equivalent to placing the following at the top of each source file:

```
#define symbol [value]
```

-Dsymbol is equivalent to -Dsymbol=1

Undefine Preprocessor Symbol

Undefines a preprocessor *symbol*. The equivalent driver option is:

- **-Usymbol**

This is equivalent to placing the following at the top of each source file:

```
#undef symbol
```

Display includes Preprocessor Directives Listing

Controls the display to standard error of a list of files opened by a `#include` directive during normal compilation. Files are compiled and linked normally.

Permitted settings for this option are:

- **On (-H)**

- **Off** (`--no_trace_includes`) — [default]

C/C++ Compiler Options

This is a top-level option category.

These options control various aspects of compilation that relate directly to the C and C++ languages.

For additional, more specialized options, see “C/C++ Compiler Options” on page 303.

C Language Dialect

Controls the version of C to be accepted by the compiler. Permitted settings for this option are:

- **Strict ISO C99 (-C99)** — Specifies strict ISO C99. This mode is compliant with ISO/IEC 9899:1999 and does not allow any non-standard constructs.
- **ISO C99 (-c99)** — Specifies ISO C99. This mode provides all features of the C99 language in addition to some extensions. Some non-standard constructs produce warnings rather than errors.
- **Strict ANSI C (-ANSI)** — Specifies strict ANSI. This mode is compliant with the ANSI X3.159-1989 standard and does not allow any non-standard constructs.
- **ANSI C (-ansi)** — [default] Specifies ANSI C with extensions. This mode extends the ANSI X3.159-1989 standard with certain useful and compatible constructs. For details, see “ANSI C” on page 565.

For details about Green Hills C and the differences between the various modes, see Chapter 14, “Green Hills C”.

- **GNU C (-gcc)** — Specifies full GNU mode, supporting GNU features (including zero size arrays, multi-line string constants, and `inline` functions) which are less compatible with ANSI C. For details, see “GNU C” on page 573.
- **K+R C (-k+r)** — Support for K&R C is deprecated and may be removed in future versions of MULTI. Specifies the C version documented in *Kernighan & Ritchie, First Edition*, which is somewhat compatible with the portable C compiler (PCC). In this mode, the compiler will accept ANSI-style prototypes. For details, see “K&R C” on page 582.

C Japanese Automotive Extensions

Controls support for the Japanese Automotive C extensions. Permitted settings for this option are:

- **On** (`-japanese_automotive_c`)
- **Off** (`-no_japanese_automotive_c`) — [default]

For more information, see “Japanese Automotive C Extensions” on page 587.

C++ Language Dialect

Specifies the version of C++ to be accepted by the compiler. Permitted settings for this option are:

- **Standard C++ (Violations Give Errors) (--STD)** — Specifies C++ Strict Standard Mode, which gives errors when non-Standard features are used and disables features that conflict with Standard C++. For more information, see “Standard C++” on page 624.
- **Standard C++ (Violations Give Warnings) (--std)** — [default] Specifies C++ Standard Mode, which gives warnings when non-Standard features are used and disables features that conflict with Standard C++. For more information, see “Standard C++” on page 624.
- **Standard C++ with ARM Extensions (--arm)** — Specifies Standard C++ with extensions, which extends the standard with many useful and compatible constructs. This dialect is not supported on INTEGRITY projects. For more information, see “Standard C++ with ARM Extensions” on page 630.
- **GNU C++ (--g++)** — Specifies GNU C++ mode. This dialect is not supported on INTEGRITY projects. For more information, see “GNU C++” on page 631.
- **Embedded C++ (--e)** — Specifies Embedded C++. For more information, see “Embedded C++” on page 627.
- **Extended Embedded C++ (--ee)** — Specifies Extended Embedded C++, which adds templates, namespaces, mutable new-style casts, and the Standard Template Library (STL) to Embedded C++. For more information, see “Extended Embedded C++” on page 629.

C++ Libraries

Specifies the type of C++ libraries to use. Permitted settings for this option are:

- **Standard C++ Library without Exceptions (--stdl)**
- **Standard C++ Library with Exceptions (--stdle)**
- **Extended Embedded C++ Library without Exceptions (--eel)**
- **Extended Embedded C++ Library with Exceptions (--eele)**
- **Embedded C++ Library without Exceptions (--el)**
- **Embedded C++ Library with Exceptions (--ele)**

The default is to use libraries equivalent to your **C/C++ Compiler**→**C++ Language Dialect** without exceptions, and you may not specify a library that is more full-featured than your **C/C++ Compiler**→**C++ Language Dialect**. For more information, see “Specifying C++ Libraries” on page 623.

C++ Exception Handling

Controls support for exception handling. Permitted settings for this option are:

- **On (--exceptions)** — Enables support for exception handling. Code size and speed may be impacted even when exception handling is not directly used.
- **Off (--no_exceptions)** — [default] Disables support for exception handling.

Allow C++ Style Slash Comments in C

Controls treatment of C++ style `//` comments. Permitted settings for this option are:

- **On (--slash_comment)** — [default] C++ style `//` comments are accepted.
- **Off (--no_slash_comment)** — C++ style `//` comments are not accepted and generate errors.

ANSI Aliasing Rules

Controls assumptions based on ANSI aliasing rules in the compiler. Permitted settings for this option are:

- **On** (**-ansi_alias**) — [default] Enables ANSI aliasing rules.
- **Off** (**-no_ansi_alias**) — Disables ANSI aliasing.

When this option is enabled, the compiler assumes that memory may only be accessed through pointers of compatible types. This allows the compilers to generate better code. Illegal casts may cause the compiler to generate code that does not behave as expected. Consider the following example:

```
void func(float *fp, int n)
{
    int *ip = (int *)fp;
    *fp = n;
    *ip |= 0x80000000; /* illegal access */
}
```

This may behave unexpectedly because an `int` type may not be used to access an object of the `float` type.

Disabling this option suppresses diagnostics 1518, 1519, and 1798.

MISRA C 2004

This option category is contained within **C/C++ Compiler**.

The MISRA C 2004 Standard was created by the Motor Industry Software Reliability Association (MISRA). It is a set of guidelines for the C programming language that is designed to enforce good practices in the development of embedded automotive systems. You can find complete details in the *Guidelines For The Use Of The C Language In Critical Systems, Motor Industry Software Reliability Association, October 2004* book. For more information, see the MISRA Web site (<http://www.misra.org.uk/>).

There are 141 rules, classified into two categories, as follows:

- 121 *required* rules (designated **[R]** below) — mandatory requirements placed on the programmer that will, by default, generate an error if enabled and breached.
- 20 *advisory* rules (designated **[A]** below) — suggestions to the programmer that will, by default, generate a warning if enabled and breached.

The MISRA rules are organized into 21 groups, each focusing on a particular C language topic (for example: types, functions, expressions, the C preprocessor,

etc.). For information about each group and individual rules, see “MISRA C 2004” on page 197.

Enforcing MISRA 2004 Rules

A single driver option allows you to enable any or all of the MISRA rules. It may be used in any of the following forms (where *n* is a rule number):

- **--misra_2004=all** — Enables checking of all the rules.
- **--misra_2004=none** — Disables checking of all the rules.
- **--misra_2004=*n*[,*n1*...]** — Enables checking of a comma-separated list of rules.
- **--misra_2004=-*n*[,-*n1*...]** — Disables checking of a comma-separated list of rules.
- **--misra_2004=*n*-*n1*** — Enables checking of rules *n* through *n1*.
- **--misra_2004=-*n*-*n1*** — Disables checking of rules *n* through *n1*.

For a list of all supported MISRA C 2004 rules, see “MISRA C 2004” on page 197.

The following example demonstrates the use of these MISRA options:

Example 1. Using MISRA Builder and Driver Options

--misra_2004=all,-2.3 — Enables checking of all rules except rule 2.3.

--misra_2004=2.3-4.2 — Enables checking of rules 2.3 through 4.2.

--misra_2004=2.3,4.2 — Enables checking of rules 2.3 and 4.2.

--misra_2004=-2.3,-4.2 — Disables checking of rules 2.3 and 4.2.

MISRA C 2004 Implementation

The Green Hills tools provide options to control the severity of messages that alert you to violations of MISRA C 2004 rules. Most of the messages are given at compile time, although some are generated at run-time. Because run-time checking can adversely affect code performance, an option is provided to disable it.

Most of the rules that generate messages at compile time are implemented though compiler checks that are only enabled when the corresponding MISRA rule is enabled. Some of the rules specify constraints that the compiler already has checks for. In these cases, the severity of the diagnostic message will be elevated to the severity specified for the corresponding MISRA rule if it exceeds the default severity for that message.

In some cases a MISRA rule has been implemented using a number of distinct compile time checks that each generate unique diagnostic messages when they are violated. This means that one item that violates one MISRA rule might cause multiple diagnostics to be issued if it violates more than one of the checks used to implement the rule. The advantage of this approach is that by modifying the severity of or suppressing individual diagnostic messages, the user can allow a deviation from a portion of a MISRA rule without disabling the entire rule. For example, if you have **Rule 8.1** enabled but want to allow static functions to be defined without a separate prototype declaration, you can specify **--diag_remark=1828** or **--diag_suppress=1828** to disable the portion of **Rule 8.1** that requires static function definitions to have separate prototype declarations.

In the introduction to section 6.15, "Switch statements," MISRA-C:2004 defines a restricted switch statement syntax. Some of the syntax restrictions are enforced as part of the rules contained in section 6.15. The remainder of the syntax restrictions are enforced by the compiler by default if any of the rules from section 6.15, 15.1-15.5, are enabled. Because all of the rules in section 6.15 are required rules, the diagnostic severity for the restricted switch statement syntax is set to the required rule severity level by default. The compiler diagnostics issued for these syntax restrictions include 1705, 1706, 1708, 1709, and 1821.

The compiler toolchain enforces some MISRA rules by disabling features or changing logic that is not specific to MISRA rule checking. If you enable checking for one of these rules, and your code violates that rule, you will receive an error regardless of the specified MISRA diagnostic level. Because the compiler toolchain does not issue the error due to the MISRA rule directly, the error message will not contain a reference to that rule. These MISRA rules are not affected by `#pragma ghs startnomisra` or `#pragma ghs endnomisra` (see "Green Hills Extension Pragma Directives" on page 683).

For example, you might enable checking for MISRA required rule 1.1, which enables strict ANSI C by disabling recognition of certain non-standard keywords. Because the compiler can no longer recognize those keywords, it

cannot parse code that violates the rule. As a result, the compiler will issue a non-discretionary error even if you have used `--misra_req=silent` to downgrade and hide MISRA required rule errors.

Some MISRA rules are enforced through preprocessor directives in the standard headers provided by Green Hills and are only enforced when the standard headers are used.

Some MISRA rules change other command line controllable options as documented in the following section. The MISRA rules override explicit settings of the other command line options, even if these options appear later on the command line than the MISRA option.

MISRA C 2004 Rules - 1 Environment

- **1.1 [R] ISO 9899:1990 C conformance w/o extensions (`--misra_2004=1.1`)** — Enables Strict ANSI C. This option enables `-ANSI`, which disables recognition of many non-standard keywords and extensions. Errors issued because of this check are not identified as MISRA errors, and their severities are unaffected by MISRA classification or the `--misra_req` option.
- **1.2 [R] No dependencies can be placed on undefined or unspecified behavior (`--misra_2004=1.2`)** — Not enforced.
- **1.3 [R] Code other than C must conform to the standard interface (`--misra_2004=1.3`)** — Not enforced.
- **1.4 [R] Compiler and linker verified to support 31 significant case-sensitive characters for identifiers (`--misra_2004=1.4`)** — Green Hills tools have no set limit by default.
- **1.5 [A] Floating-point implementation is standard-compliant (`--misra_2004=1.5`)** — The Green Hills compilers use the IEEE floating-point representation. This rule cannot be disabled.

MISRA C 2004 Rules - 2 Language Extensions

- **2.1 [R] Inline assembly only in functions or macros with no other code (`--misra_2004=2.1`)** — Enforced.

Associated diagnostics include 1749 and 1839.

- **2.2 [R] Use of `/*...*/` comments only (`--misra_2004=2.2`)** — Enforced.

Associated diagnostics include 1876.

- **2.3 [R] No nested comments (--misra_2004=2.3)** — Enforced.

Associated diagnostics include 9.

- **2.4 [A] No ‘commented out’ sections of code (--misra_2004=2.4)** — Does not allow the following characters inside of a comment:
 - A semicolon followed by by a new-line character.
 - Open or closed curly braces.

Associated diagnostics include 1771.

MISRA C 2004 Rules - 3 Documentation

- **3.1 [R] Documentation of implementation-defined behavior required (--misra_2004=3.1)** — Not enforced.
- **3.2 [R] Values of char types restricted to subset of ISO 10646-1 (--misra_2004=3.2)** — All Green Hills compilers use the ASCII standard for mapping character sets to numeric values. This property cannot be disabled.
- **3.3 [A] Implementation of integer division determined and documented (--misra_2004=3.3)** — This behavior depends on the hardware configuration of the target. Most Green Hills compilers, however, round the division result toward 0 (including PowerPC, MIPS, TriCore, and x86).
- **3.4 [R] Uses of #pragma documented and explained (--misra_2004=3.4)** — This book lists all #pragma constructs, their usages, and effects (see “General Pragma Directives” on page 680).
- **3.5 [R] Documentation of implementation-defined behavior and packing of bitfields required (--misra_2004=3.5)** — Not enforced.
- **3.6 [R] All library production code conforms to MISRA (--misra_2004=3.6)** — Not enforced.

MISRA C 2004 Rules - 4 Character Sets

- **4.1 [R] Only ISO C escape sequences used (--misra_2004=4.1)** — Enforced.

Associated diagnostics include 192 and 1823.

- **4.2 [R] No trigraphs (--misra_2004=4.2)** — Enforced.

Associated diagnostics include 1695.

MISRA C 2004 Rules - 5 Identifiers

- **5.1 [R] No more than 31 chars to determine an identifier (--misra_2004=5.1)** — Enforced by truncating all identifiers at 31 characters.

Errors issued because of this check are not identified as MISRA errors, and their severities are unaffected by MISRA classification or the **--misra_req** option.

Associated diagnostics include 1772.

- **5.2 [R] No use of same id name in inner and outer scope (--misra_2004=5.2)** — Enforced.

Associated diagnostics include 460 and 1721.

- **5.3 [R] Each ‘typedef’ must be a unique identifier (--misra_2004=5.3)** — Enforced within a translation unit.

Associated diagnostics include 1722 and 1841.

- **5.4 [R] Tag names must be unique (--misra_2004=5.4)** — Enforced within a translation unit.

Associated diagnostics include 188, 469, and 1842.

- **5.5 [R] No reuse of function or static object identifiers (--misra_2004=5.5)** — Enforced within a translation unit.

Associated diagnostics include 1843.

- **5.6 [A] No identifiers with the same name in different name spaces except for struct and union members (--misra_2004=5.6)** — Enforced within a translation unit.

Associated diagnostics include 1723.

- **5.7 [A] No reuse of identifiers (--misra_2004=5.7)** — Enforced within a translation unit.

Associated diagnostics include 1840.

MISRA C 2004 Rules - 6 Types

- **6.1 [R] ‘char’ only used for storage and use of character values**
(--misra_2004=6.1) — Enforced.

Associated diagnostics include 1850 and 1852.

- **6.2 [R] ‘signed char’ and ‘unsigned char’ only used for storage and use of numeric values** (--misra_2004=6.2) — Enforced.

Associated diagnostics include 1851.

- **6.3 [A] Basic types used only in ‘typedef’s and bitfields**
(--misra_2004=6.3) — Enforced.

Associated diagnostics include 1698.

- **6.4 [R] Bitfields can only have ‘unsigned int’ or ‘signed int’ types**
(--misra_2004=6.4) — Enforced.

Associated diagnostics include 230 and 1717.

- **6.5 [R] Signed bitfields must be least 2 bits long** (--misra_2004=6.5) — Enforced.

Associated diagnostics include 108.

MISRA C 2004 Rules - 7 Constants

- **7.1 [R] No non-zero octal constants or octal escape sequences**
(--misra_2004=7.1) — Enforced.

Associated diagnostics include 1718.

MISRA C 2004 Rules - 8 Declarations and Definitions

- **8.1 [R] Functions must always have prototype declarations**
(--misra_2004=8.1) — Enforced.

Associated diagnostics include 223, 1547, 1791, 1800, and 1828.

- **8.2 [R] Every function must have an explicit return type**
(--misra_2004=8.2) — Enforced.

Associated diagnostics include 77, 260, and 938.

- **8.3 [A] Function declaration and definition prototypes match**
(`--misra_2004=8.3`) — Enforced.

Associated diagnostics include 1844.

- **8.4 [R] Multiple declarations of an object or function must be compatible**
(`--misra_2004=8.4`) — Enforced with a translation unit.

Associated diagnostics include 147.

- **8.5 [R] No object or function definitions in a header file**
(`--misra_2004=8.5`) — Enforced.

Associated diagnostics include 1846.

- **8.6 [R] Functions always declared at file scope** (`--misra_2004=8.6`) — Enforced.

Associated diagnostics include 1731.

- **8.7 [R] Use function or block scope definitions for objects whenever possible** (`--misra_2004=8.7`) — Enforced for non-static variables in `elxr`. If you enable `-auto_sda` or use the `__thread` keyword, this rule might not be enforced.

- **8.8 [R] External objects and functions declared in no more than one file**
(`--misra_2004=8.8`) — Not enforced.

- **8.9 [R] Only one external definition for an external identifier**
(`--misra_2004=8.9`) — Sets `--no_commons`, which prevents multiple definitions of uninitialized global variables. Linker errors issued because of this check are not identified as MISRA errors, and their severity is unaffected by MISRA classification or the `--misra_req` option.

Associated diagnostics include 1797.

- **8.10 [R] Static linkage of file scope declarations when possible**
(`--misra_2004=8.10`) — Enforced in `elxr`. If you use the `__thread` keyword, this rule might not be enforced.
- **8.11 [R] All objects and functions with internal linkage declared ‘static’**
(`--misra_2004=8.11`) — Enforced.

Associated diagnostics include 172.

- **8.12 [R] Arrays with external linkage must have known size at compile time** (`--misra_2004=8.12`) — Enforced.

Associated diagnostics include 1824.

MISRA C 2004 Rules - 9 Initialization

- **9.1 [R] Automatic variables initialized before used (--misra_2004=9.1)** — Enforced for simple cases.

Associated diagnostics include 549.

- **9.2 [R] Braces used to match structure in initialization of arrays or structs (--misra_2004=9.2)** — Enforced.

Associated diagnostics include 146, 991, and 1736.

- **9.3 [R] All or only first member of an enumeration may be explicitly initialized (--misra_2004=9.3)** — Enforced.

Associated diagnostics include 1726.

MISRA C 2004 Rules - 10 Arithmetic Type Conversions

- **10.1 [R] Restrict implicit conversions for integer type expressions (--misra_2004=10.1)** — Enforced.

Associated diagnostics include 1863, 1864, 1865, 1866, 1867, 1868, 1869, and 1870.

- **10.2 [R] Restrict implicit conversions for floating type expressions (--misra_2004=10.2)** — Enforced.

Associated diagnostics include 1871, 1872, 1873, 1874, and 1875.

- **10.3 [R] Restrict explicit casts for integer type expressions (--misra_2004=10.3)** — Enforced.

Associated diagnostics include 1847, 1878, 1879, and 1880.

- **10.4 [R] Restrict explicit casts for floating type expressions (--misra_2004=10.4)** — Enforced.

Associated diagnostics include 1848.

- **10.5 [R] Bitwise ‘~’ and ‘<<’ expressions on unsigned char or unsigned short types must be cast to underlying type (--misra_2004=10.5)** — Enforced.

Associated diagnostics include 1849.

- **10.6 [R] Apply ‘U’ suffix to all constants of ‘unsigned’ type** (`--misra_2004=10.6`) — Enforced.

Associated diagnostics include 1775.

MISRA C 2004 Rules - 11 Pointer Type Conversions

- **11.1 [R] No conversions between pointer to function and non-integral types** (`--misra_2004=11.1`) — Enforced.

Associated diagnostics include 1832.

- **11.2 [R] No conversions between pointer to object and any type other than integral, pointer to object, or pointer to void** (`--misra_2004=11.2`) — Enforced.

Associated diagnostics include 1833.

- **11.3 [A] No casting between pointer and integral types** (`--misra_2004=11.3`) — Enforced.

Associated diagnostics include 1834 and 1877.

- **11.4 [A] No casting between different pointer to object types** (`--misra_2004=11.4`) — Enforced.

Associated diagnostics include 1835.

- **11.5 [R] No casting that removes any ‘const’ or ‘volatile’ qualification from the type addressed by a pointer** (`--misra_2004=11.5`) — Enforced.

Associated diagnostics include 1836.

MISRA C 2004 Rules - 12 Expressions

- **12.1 [A] Limited dependence on C precedence rules** (`--misra_2004=12.1`) — Enforced per the guidelines in the supporting text of the rule.

Associated diagnostics include 1737.

- **12.2 [R] No expressions with values dependent on evaluation order** (`--misra_2004=12.2`) — Not enforced.

- **12.3 [R] No side effects in the operand of ‘sizeof’ (--misra_2004=12.3)**
— Enforced.

Associated diagnostics include 1725.

- **12.4 [R] No side effects in the right hand operand of ‘&&’ or ‘||’**
(--misra_2004=12.4) — Enforced.

Associated diagnostics include 1724.

- **12.5 [R] Operands of ‘&&’ and ‘||’ must be primary expressions**
(--misra_2004=12.5) — Enforced.

Associated diagnostics include 1729.

- **12.6 [A] Operands of logical operators must be Boolean, and Boolean expressions may not be used as operands of other operators**
(--misra_2004=12.6) — Enforced.

Associated diagnostics include 1853 and 1854.

- **12.7 [R] No bitwise operations on signed integer types**
(--misra_2004=12.7) — Enforced.

Associated diagnostics include 1730.

- **12.8 [R] Value of right hand operand of a shift operator must be equal to or greater than zero and less than the size of the underlying type of the left hand operand (--misra_2004=12.8)** — Enforced. Both compile-time and run-time checks are performed. The run-time check is disabled when the --no_misra_runtime option is passed.

Associated diagnostics include 62 and 63.

- **12.9 [R] No unary minus on unsigned expressions (--misra_2004=12.9)**
— Enforced.

Associated diagnostics include 1702.

- **12.10 [R] No comma operators (--misra_2004=12.10)** — Enforced.

Associated diagnostics include 1825.

- **12.11 [A] No wrap-around in constant unsigned expression evaluation**
(--misra_2004=12.11) — Enforced.

Associated diagnostics include 1735.

- **12.12 [R] No use of underlying bit representation in floating point expressions (--misra_2004=12.12)** — The Green Hills compilers do not allow any use of bit-wise operators (such as &, |, ^, ~, etc.) by default. When this rule is enabled, a diagnostic is issued for union members of floating-point type.

Associated diagnostics include 1820.

- **12.13 [A] No mixing of increment (++) and decrement (--) operators with other operators (--misra_2004=12.13)** — Enforced.

Associated diagnostics include 1856.

MISRA C 2004 Rules - 13 Control Statement Expressions

- **13.1 [R] Assignment operators not used in Boolean expressions (--misra_2004=13.1)** — Enforced.

Associated diagnostics include 1738.

- **13.2 [A] Explicit test of a value against zero unless the expression is Boolean (--misra_2004=13.2)** — Enforced.

Associated diagnostics include 1855 and 1881.

- **13.3 [R] Floating-point values not tested for (in)equality (--misra_2004=13.3)** — Enforced.

Associated diagnostics include 1704.

- **13.4 [R] No floating-point variables in ‘for’ loop control expressions (--misra_2004=13.4)** — Enforced.

Associated diagnostics include 1750.

- **13.5 [R] Only loop control expression in ‘for’ statement header (--misra_2004=13.5)** — Not enforced.

- **13.6 [R] No modification of numeric control variables in ‘for’ loop body (--misra_2004=13.6)** — Not enforced.

- **13.7 [R] No Boolean operations with invariant results (--misra_2004=13.7)** — Enforced.

Associated diagnostics include 1857.

MISRA C 2004 Rules - 14 Control Flow

- **14.1 [R] No unreachable code (--misra_2004=14.1) — Enforced.**

Associated diagnostics include 111 and 177.

- **14.2 [R] All non-null statements must have a side-effect (--misra_2004=14.2) — Enforced.**

Associated diagnostics include 174.

- **14.3 [R] Null statement must occur on a line by itself (--misra_2004=14.3) — Enforced.**

Associated diagnostics include 1746.

- **14.4 [R] No ‘goto’ statements (--misra_2004=14.4) — Enforced.**

Associated diagnostics include 1705.

- **14.5 [R] No ‘continue’ statements (--misra_2004=14.5) — Enforced.**

Associated diagnostics include 1706.

- **14.6 [R] At most one break statement per iteration statement (--misra_2004=14.6) — Enforced.**

Associated diagnostics include 1845.

- **14.7 [R] Functions must have a single point of exit (--misra_2004=14.7) — Enforced.**

Associated diagnostics include 1734.

- **14.8 [R] Dependent statements of loop and switch statements must have braces (--misra_2004=14.8) — Enforced.**

Associated diagnostics include 1709.

- **14.9 [R] Dependent statements of if statements must have braces (--misra_2004=14.9) — Enforced.**

Associated diagnostics include 1826.

- **14.10 [R] All ‘if’...‘else if’ constructs must have an ‘else’ (--misra_2004=14.10) — Enforced.**

Associated diagnostics include 1710.

MISRA C 2004 Rules - 15 Switch Statements

- **15.1 [R] Switch labels must be at top level compound statement of ‘switch’ statement (--misra_2004=15.1) — Enforced.**

Associated diagnostics include 1838.

- **15.2 [R] Every non-empty switch clause terminates with an unconditional ‘break’ statement (--misra_2004=15.2) — Enforced.**

Associated diagnostics include 1711 and 1884.

- **15.3 [R] Every ‘switch’ statement must contain a final ‘default’ clause (--misra_2004=15.3) — Enforced.**

Associated diagnostics include 1712 and 1837.

- **15.4 [R] No Boolean values in ‘switch’ expressions (--misra_2004=15.4) — Enforced.**

Associated diagnostics include 1733.

- **15.5 [R] ‘switch’ statements must have at least one ‘case’ (--misra_2004=15.5) — Enforced.**

Associated diagnostics include 1713.

MISRA C 2004 Rules - 16 Functions

- **16.1 [R] No variable argument function definitions (--misra_2004=16.1) — Enforced.**

Associated diagnostics include 1697.

- **16.2 [R] No direct or indirect recursion (--misra_2004=16.2) — Enforced** in the `elxr` linker. Indiscriminate use of function pointers might cause the linker to miss certain recursive functions, or report errors for nonrecursive functions. Coverage Profiling may also cause nonrecursive functions to be reported incorrectly as recursive (see “**Profiling - Coverage**” on page 188).

- **16.3 [R] Identifiers must be given for all function parameters (--misra_2004=16.3) — Enforced.**

Associated diagnostics include 1727.

- **16.4 [R] Parameter names in function declaration and definition must match (--misra_2004=16.4) — Enforced.**

Associated diagnostics include 1745.

- **16.5 [R] Declarations of functions with no parameters must have a ‘void’ parameter (--misra_2004=16.5) — Enforced.**

Associated diagnostics include 1728.

- **16.6 [R] Number of arguments must match the function prototype (--misra_2004=16.6) — Always enforced.**
- **16.7 [A] Pointer parameters to functions declared as pointer to ‘const’ if possible (--misra_2004=16.7) — Not enforced.**
- **16.8 [R] Return expression must match function type (--misra_2004=16.8) — Enforced.**

Associated diagnostics include 117 and 940.

- **16.9 [R] Function identifiers may only be used for calls or with preceding ‘&’ operator (--misra_2004=16.9) — Enforced.**

Associated diagnostics include 1774.

- **16.10 [R] If error information returned by a function, it must be tested (--misra_2004=16.10) — Not enforced.**

MISRA C 2004 Rules - 17 Pointers and Arrays

- **17.1 [R] No pointer arithmetic on pointers that don’t address an array element (--misra_2004=17.1) — Not enforced.**
- **17.2 [R] No pointer subtraction on pointers that don’t address elements of the same array (--misra_2004=17.2) — Not enforced.**
- **17.3 [R] Relational operators not used on pointers that don’t point to the same array (--misra_2004=17.3) — Not enforced.**
- **17.4 [R] No pointer arithmetic other than array indexing (--misra_2004=17.4) — Enforced.**

Associated diagnostics include 1752 and 1858.

- **17.5 [A] No more than 2 levels of pointer indirection in an object declaration (--misra_2004=17.5) — Enforced.**

Associated diagnostics include 1741.

- **17.6 [R] Addresss of automatic variable not used out of scope** (`--misra_2004=17.6`) — Enforced.

Associated diagnostics include 1056 and 1780.

MISRA C 2004 Rules - 18 Structs and Unions

- **18.1 [R] No incomplete struct or union types at end of translation unit** (`--misra_2004=18.1`) — Not enforced.
- **18.2 [R] No assignments between overlapping objects** (`--misra_2004=18.2`) — Not enforced.
- **18.3 [R] No reuse of memory for unrelated purposes** (`--misra_2004=18.3`) — Not enforced.
- **18.4 [R] Unions may not be used** (`--misra_2004=18.4`) — Enforced.

Associated diagnostics include 1827.

MISRA C 2004 Rules - 19 Preprocessing Directives

- **19.1 [A] Only preprocessing directives and comments before ‘#include’** (`--misra_2004=19.1`) — Enforced.

Associated diagnostics include 1748.

- **19.2 [A] Only standard characters in file names for ‘#include’** (`--misra_2004=19.2`) — Enforced.

Associated diagnostics include 1747.

- **19.3 [R] ‘#include’ directive only followed by <filename> or "filename"** (`--misra_2004=19.3`) — Always enforced.
- **19.4 [R] Restrict macro syntax** (`--misra_2004=19.4`) — Enforced.

Associated diagnostics include 1859, 1860, and 1886.

- **19.5 [R] No ‘#define’ or ‘#undef’ within a block** (`--misra_2004=19.5`) — Enforced.

Associated diagnostics include 1732.

- **19.6 [R] ‘#undef’ cannot be used (--misra_2004=19.6) — Enforced.**

Associated diagnostics include 1715.

- **19.7 [A] Function used instead of function-like macro when possible (--misra_2004=19.7) — Enforced.**

Associated diagnostics include 1862.

- **19.8 [R] Function-like macro must be called with all of its arguments (--misra_2004=19.8) — Enforced.**

Associated diagnostics include 54 and 76.

- **19.9 [R] No preprocessing directives in function-like macros (--misra_2004=19.9) — Enforced.**

Associated diagnostics include 10.

- **19.10 [R] In function-like macro definition, wrap each parameter reference in parentheses (--misra_2004=19.10) — Enforced.**

Associated diagnostics include 1829.

- **19.11 [R] Identifiers in preprocessing directives defined before used (--misra_2004=19.11) — Enforced.**

Associated diagnostics include 193.

- **19.12 [R] At most one ‘#’ or ‘##’ operator in a macro (--misra_2004=19.12) — Enforced.**

Associated diagnostics include 1716.

- **19.13 [A] No ‘#’ or ‘##’ preprocessor operators (--misra_2004=19.13) — Enforced.**

Associated diagnostics include 1830.

- **19.14 [R] Correct use of ‘defined’ preprocessor operator (--misra_2004=19.14) — Enforced.**

Associated diagnostics include 1831.

- **19.15 [R] Prevent contents of header files from being included more than once (--misra_2004=19.15) — A diagnostic is issued if the front end does not recognize an #ifndef...#define...#endif include guard.**

Because **assert.h** does not conform to this rule, a diagnostic is issued if you include this header file and enable this rule.

Associated diagnostics include 1882.

- **19.16 [R] Conditionally excluded preprocessor directives must be syntactically valid (--misra_2004=19.16) — Enforced.**

Associated diagnostics include 11.

- **19.17 [R] All related conditional preprocessor directives must reside in the same file (--misra_2004=19.17) — Enforced.**

Associated diagnostics include 36 and 37.

MISRA C 2004 Rules - 20 Standard Libraries

The following MISRA rules are enforced both by `#error` directives in the standard library headers, and by diagnostics emitted by the `elxr` linker. The linker diagnostics prevent programs from circumventing these rules by calling forbidden functions without including the standard headers (after suppressing MISRA 2004 Rule 8.1). The linker prevents any use of the forbidden functions in any library modules that are pulled into the link.

- **20.1 [R] No definition, redefinition, or undefinition of reserved words and standard library names (--misra_2004=20.1) — Enforced.**

Associated diagnostics include 45, 46, 1861, and 1885.

- **20.2 [R] Standard library macro, object, and function names cannot be reused (--misra_2004=20.2) — Not enforced.**
- **20.3 [R] Check validity of values passed to library functions (--misra_2004=20.3) — Not enforced.**
- **20.4 [R] Dynamic heap memory allocation cannot be used (--misra_2004=20.4) — Enforced.**
- **20.5 [R] The error indicator 'errno' cannot be used (--misra_2004=20.5) — Enforced.**
- **20.6 [R] Macro 'offsetof' in <stddef.h> cannot be used (--misra_2004=20.6) — Enforced.**
- **20.7 [R] The 'setjmp' and 'longjmp' facilities cannot be used (--misra_2004=20.7) — Enforced.** Because the debugging memory library

libdbmem.a uses `setjmp()`, building an otherwise conforming program with **-check=alloc** might trigger the associated **elxr** diagnostic.

- **20.8 [R] The facilities of <signal.h> cannot be used (--misra_2004=20.8)** — Enforced.
- **20.9 [R] No use of <stdio.h> in production code (--misra_2004=20.9)** — Enforced.
- **20.10 [R] Functions ‘atof’, ‘atoi’, and ‘atol’ are not used (--misra_2004=20.10)** — Enforced.
- **20.11 [R] Functions ‘abort’, ‘exit’, ‘getenv’, and ‘system’ are not used (--misra_2004=20.11)** — Enforced. Because the default startup code uses `exit()`, an otherwise conforming program might trigger the associated **elxr** diagnostic.
- **20.12 [R] The facilities of <time.h> cannot be used (--misra_2004=20.12)** — Enforced.

MISRA C 2004 Rules - 21 Run-time Failures

- **21.1 [R] Run-time checking (--misra_2004=21.1)** — Enforced by performing run-time error checks for array bounds, assignment bounds, NULL pointer dereference, divide-by-zero, and unresolved switch statement condition. For more information about run-time error checking, see “**Run-Time Error Checks**” on page 188. The **--no_misra_runtime** option disables this rule.

MISRA C - Required Rules Level

Controls the diagnostic messages generated upon a violation of the MISRA *required* rules. Permitted settings for this option are:

- **Errors (--misra_req=error)** — [default]
- **Warnings (--misra_req=warn)**
- **Silent (--misra_req=silent)**

MISRA C - Advisory Rules Level

Controls the diagnostic messages generated upon a violation of the MISRA *advisory* rules. Permitted settings for this option are:

- **Errors** (`--misra_adv=error`)
- **Warnings** (`--misra_adv=warn`) — [default]
- **Silent** (`--misra_adv=silent`)

MISRA C - Run-Time Checks

Controls the performance of run-time checks for the MISRA rules that require them. Permitted settings for this option are:

- **On** (`--misra_runtime`) — [default]
- **Off** (`--no_misra_runtime`)

MISRA C 1998

This option category is contained within **C/C++ Compiler**.

This section contains options that allow you to check your code for compliance with the Motor Industry Software Reliability Association (MISRA) rules, a set of guidelines for the C programming language that is designed to enforce good practices in the development of embedded automotive systems.



Note Support for MISRA 1998 Rules is deprecated and may be removed in future versions of **MULTI**. It is recommended that you use the MISRA 2004 Rules (see “MISRA C 2004” on page 197).

MISRA C 1998 Rules - Environment

Permitted settings for this option are:

- **1. [R] ISO 9899 C conformance w/o extensions** — Enables Strict ANSI C. This option enables `-ANSI`, which disables recognition of many non-standard keywords and extensions. Errors issued because of this check are not identified as MISRA errors, and their severities are unaffected by MISRA classification or the `--misra_req` option.
- **2. [A] Code other than C must conform to standard interface** — Not enforced.
- **3. [A] Inline assembly only in functions with no other code**

- **4. [A] Run-time checking** — Implemented through run-time checking. This option sets **-check=assignbound**, **-check=bound**, **-check=nilderef**, **-check=zerodivide**, and **-check=switch**. For more information about run-time error checking, see “Run-Time Error Checks” on page 188. The option **--no_misra_runtime** turns off this rule completely. The option **--no_misra_runtime** disables both this rule and rule 107.

MISRA C 1998 Rules - Character Set

Permitted settings for this option are:

- **5. [R] Only ISO C characters and escape sequences used** — Disables recognition of the `\e` escape in GNU mode. Issues a MISRA diagnostic on each use of a multibyte character literal (such as a Kanji character).
- **6. [R] Values of char types restricted to subset of ISO 10646-1** — All Green Hills compilers use the ASCII standard for mapping character sets to numeric values. This property cannot be turned off.
- **7. [R] No trigraphs**
- **8. [R] No multibyte chars and wide strings** — Issues a MISRA diagnostic on each use of a multibyte character literal (such as a Kanji character).

MISRA C 1998 Rules - Comments

Permitted settings for this option are:

- **9. [R] No nested comments**
- **10. [A] No 'commented out' sections of code** — Disallows the following characters inside of a comment:
 - semicolon followed by a new-line character
 - open or close curly braces

MISRA C 1998 Rules - Identifiers

Permitted settings for this option are:

- **11. [R] No more than 31 chars to determine an identifier** — Enforced by truncating all identifiers at 31 characters.

Errors issued because of this check are not identified as MISRA errors, and their severities are unaffected by MISRA classification or the `--misra_req` option.

Associated diagnostics include 1772.

- **12. [A] No identifiers with the same names in different namespaces**

MISRA C 1998 Rules - Types

Permitted settings for this option are:

- **13. [A] Basic types used only in ‘typedef’s**
- **14. [R] ‘char’ always used as ‘signed char’ or ‘unsigned char’**
- **15. [A] Floating point implementation complies to a standard** — The Green Hills compilers use the IEEE floating-point representation. This rule cannot be turned off.
- **16. [R] No underlying use of bits in floating point expressions** — The Green Hills compilers disallow any use of bit-wise operators (such as `&`, `|`, `^`, `~`, etc.) by default. When this rule is enabled, a diagnostic is issued for union members of floating-point type.
- **17. [R] ‘typedef’ names shall not be reused**

MISRA C 1998 Rules - Constants

Permitted settings for this option are:

- **18. [A] Numeric constants need suffixes when appropriate**
- **19. [R] No octal constants (other than zero)**

MISRA C 1998 Rules - Declarations & Definitions

Permitted settings for this option are:

- **20. [R] All objects and functions declared before used**
- **21. [R] No use of same id name in inner and outer scope**
- **22. [A] Function scope declarations whenever possible** — Not enforced.

- **23. [A] Static linkage of file scope declarations when possible** — Not enforced.
- **24. [R] No internal and external linkages of same identifiers**
- **25. [R] Only one external definition of external identifier** — Sets `--no_commons`, which prevents multiple definitions of uninitialized global variables. Linker errors issued because of this check are not identified as MISRA errors, and their severity is unaffected by MISRA classification or the `--misra_req` option.

Associated diagnostics include 1797.

- **26. [R] Compatible multiple declarations of the same object/function** — The Green Hills compilers allow only compatible declarations of multiply-declared objects and functions within the same translation unit (by default). No such checks are performed across different translation units.
- **27. [A] External objects declared in no more than one file** — Not enforced.
- **28. [A] The ‘register’ storage class specifier should not be used**
- **29. [R] The use of tag shall agree with its declaration**

MISRA C 1998 Rules - Initialization

Permitted settings for this option are:

- **30. [R] Automatic variables initialized before used** — Enforced for simple cases.
- **31. [R] Braces used in non-zero initialization of arrays/structs**
- **32. [R] All or only first enumerator may be explicitly initialized**

MISRA C 1998 Rules - Operators

Permitted settings for this option are:

- **33. [R] No side effects in right hand operand of ‘&&’ or ‘||’**
- **34. [R] Operands of ‘&&’ and ‘||’ shall be primary expressions**
- **35. [R] Assignment operators not used in Boolean expressions**

- **36. [A] Logical and bitwise operators should not be confused** — Not enforced.
- **37. [R] No bitwise operations on signed integer types**
- **38. [R] Right hand value of shift operand must be in range** — Both compile-time and run-time checks are performed. The run-time check is turned off when the option `--no_misra_runtime` is passed.
- **39. [R] No unary minus operand on unsigned expressions**
- **40. [A] No side effects in the ‘sizeof’ operand**
- **41. [A] Implementation of division determined and documented** — This behavior is dependent on the hardware configuration of the target. Most Green Hills compilers, however, round the division result toward 0 (including PowerPC, MIPS, TriCore, and x86).
- **42. [R] No comma operators except in control expr of ‘for’ loops**

MISRA C 1998 Rules - Conversions

Permitted settings for this option are:

- **43. [R] No implicit conversions which might lose information**
- **44. [A] Redundant explicit cast should not be used**
- **45. [R] Type casting to or from pointers should not be used**

MISRA C 1998 Rules - Expressions

Permitted settings for this option are:

- **46. [R] No expression with values dependent on evaluation order** — Not enforced.
- **47. [A] No dependence placed on C precedence rules**
- **48. [A] Mixed precision arithmetic must use explicit casting**
- **49. [A] Test value against zero unless expression Boolean** — Not enforced. The Green Hills C compilers do not have built-in types which would represent the Boolean type.
- **50. [R] Floating point values not tested for (in)equality**
- **51. [A] No wraparound in constant unsigned expressions**

MISRA C 1998 Rules - Control Flow

Permitted settings for this option are:

- 52. [R] There shall be no unreachable code
- 53. [R] All non-null statements shall have a side-effect
- 54. [R] Null statement must occur on line by itself
- 55. [A] No labels except in ‘switch’ statements
- 56. [R] The ‘goto’ statement shall not be used
- 57. [R] The ‘continue’ statement shall not be used
- 58. [R] No ‘break’ statement except in ‘switch’
- 59. [R] Dependent statements always enclosed in braces
- 60. [A] All ‘if’ and ‘else if’ constructs must have an ‘else’
- 61. [R] Every non-empty ‘case’ clause terminated with ‘break’
- 62. [R] All ‘switch’ statements should contain a ‘default’ clause
- 63. [A] No Boolean values in ‘switch’ expressions
- 64. [R] ‘switch’ statements need at least one ‘case’
- 65. [R] No floating-point variables in loop counters
- 66. [A] Only loop control expression in ‘for’ statement header — Not enforced.
- 67. [A] No modification of control variables in ‘for’ loop body — Not enforced.

MISRA C 1998 Rules - Functions

Permitted settings for this option are:

- 68. [R] Functions always declared at file scope
- 69. [R] No variable argument functions
- 70. [R] No direct or indirect recursion
- 71. [R] Functions always have prototype declarations This option sets `--prototype_errors` or `--prototype_warnings`, depending on the MISRA C Required Rules Level (see **Functions Witout Prototypes** in “C/C++ Messages” on page 261).

- **72. [R] Function declaration and definition match prototypes** — Always enforced.
- **73. [R] Identifiers given for all or none of function parameters**
- **74. [R] Declaration and definition match parameter names**
- **75. [R] Every function shall have an explicit return type**
- **76. [R] Functions with no parameters should have ‘void’ parameter**
- **77. [R] Unqualified arguments of callee and caller compatible** — Not enforced.
- **78. [R] Number of arguments matches the function prototype** — Always enforced.
- **79. [R] Values returned by ‘void’ functions not used** — Always enforced.
- **80. [R] No void expressions passed as parameters** — Always enforced.
- **81. [A] ‘const’ present when needed on reference parameters** — Not enforced.
- **82. [A] A function should have a single point of exit**
- **83. [R] Return expression matches function type**
- **84. [R] No expressions returned from ‘void’ functions** — Always enforced.
- **85. [A] Functions called with empty () if no parameters**
- **86. [A] If error information returned, it should be tested** — Not enforced.

MISRA C 1998 Rules - Preprocessor

Permitted settings for this option are:

- **87. [R] Only preprocessing directives before #include**
- **88. [R] Only standard characters in file names for #include**
- **89. [R] #include directive followed by <filename> or ‘filename’** — Always enforced.
- **90. [R] C macros only as constant, function-like, or specifier/qualifier**
- **91. [R] Macros not ‘#define’d or ‘#undef’d within a block**
- **92. [A] #undef should not be used**

- **93. [A] Function used instead of function-like macro when possible** — Not enforced.
- **94. [R] Function-like macro called with all of its arguments** — Always enforced.
- **95. [R] No preprocessing directives in function-like macros**
- **96. [R] Use of parentheses in function-like macros**
- **97. [A] Identifiers in pre-processing directives defined before used**
- **98. [R] Only one use of ‘#’ or ‘##’ operator in any one macro**
- **99. [R] Uses of #pragmas documented and explained** — This book lists all #pragma constructs, their usage and effects (see “Pragma Directives” on page 679).
- **100. [R] Correct use of ‘defined’ preprocessor operator** — Always enforced.

MISRA C 1998 Rules - Pointers and Arrays

Permitted settings for this option are:

- **101. [A] Pointer arithmetic should not be used**
- **102. [A] No more than 2 levels of pointer indirection**
- **103. [R] Relational operators not used except on same object** — Not enforced.
- **104. [R] No non-constant pointers to functions**
- **105. [R] Function and a pointer to it match in prototypes**
- **106. [R] Address of automatic variable not used out of scope** — The compiler checks if an address of a local variable is assigned to a global value.
- **107. [R] The NULL pointer should not be dereferenced** — Performed via a run-time check (`-check=nilderef`). For more information about run-time error checking, see “Run-Time Error Checks” on page 188. The option `--no_misra_runtime` disables both this rule and rule 4.

MISRA C 1998 Rules - Structs and Unions

Permitted settings for this option are:

- **108. [R] All members of structures/union fully specified** — Not enforced.
- **109. [R] Overlapping variable storage should not be used** — Not enforced.
- **110. [R] Unions not used to access sub-parts of larger data types** — Not enforced.
- **111. [R] Bit fields can only have ‘unsigned int’ or ‘signed int’ types**
- **112. [R] Bit fields of ‘signed int’ type must be at least 2 bits long**
- **113. [R] All members can be accessed only through their name**

MISRA C 1998 Rules - Standard Library

Permitted settings for this option are:

- **114. [R] No redefinition of reserved words and standard library names** — Always enforced. The Green Hills compilers do not allow redefinitions of C reserved words. Furthermore, the predefined macro names defined, `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__` cannot be redefined.
- **115. [R] Standard library function names shall not be reused** — Not enforced.
- **116. [R] All library production code conforms to MISRA** — Not enforced.
- **117. [R] Check validity of value passed to library functions** — Not enforced.
- **118. [R] Dynamic heap memory allocation shall not be used**
- **119. [R] The error indicator ‘errno’ shall not be used**
- **120. [R] Macro ‘offsetof’ in <stddef.h> shall not be used**
- **121. [R] <locale.h> and ‘setlocale’ function shall not be used**
- **122. [R] The ‘setjmp’ and ‘longjmp’ facilities shall not be used**
- **123. [R] The facilities of <signal.h> shall not be used**
- **124. [R] No use of <stdio.h> in production code**
- **125. [R] Functions ‘atof’, ‘atoi’, and ‘atol’ not used**
- **126. [R] Functions ‘abort’, ‘exit’, ‘getenv’, and ‘system’ not used**

- **127. [R] The facilities of `<time.h>` shall not be used**

Data Types

This option category is contained within **C/C++ Compiler**.

These options control the treatment of particular data types.

Signedness of Char Type

Specifies the signedness of the `char` type. Permitted settings for this option are:

- **Signed (`--signed_chars`)**
- **Unsigned (`--unsigned_chars`)** — [default]

Note that the standard libraries are built with the default signedness. While most library routines work the same way regardless of the signedness of the `char` type, `localeconv()` uses the default value of `CHAR_MAX`, which will not match `CHAR_MAX` in the application if the signedness is changed.

Signedness of Bitfields

Specifies the treatment of bitfields that are not explicitly declared `signed` or `unsigned`. Permitted settings for this option are:

- **Signed (`--signed_fields`)** — Multiple-bit bitfields declared with an integer type have the signedness of their declared type. Bitfields declared with an enumeration type may be either signed or unsigned to accommodate the sign and the magnitude of their enumerators.
- **Unsigned (`--unsigned_fields`)** — [default] Bitfields declared with an integer type are unsigned. Bitfields declared with an enumeration type are unsigned unless the enumeration type has a negative enumerator.

For more information, see “Signedness of Bitfields” on page 592.

Signedness of Pointers

Specifies the signedness of pointers and addresses. Permitted settings for this option are:

- **Signed** (`--signed_pointer`)
- **Unsigned** (`--unsigned_pointer`) — [default] Use this option if you have an object that spans from 0x7FFFFFFF to 0x80000000.

Use Smallest Type Possible for Enum

Controls the allocation of enumerations. Permitted settings for this option are:

- **On** (`--short_enum`) — Store enumerations in the smallest possible type.
- **Off** (`--no_short_enum`) — [default] Store enumerations as integers.

Long Long Support

Controls support for the `long long` data type. Permitted settings for this option are:

- **On** (`--long_long`) — [default] Support the `long long` data type.
- **Off** (`--no_long_long`) — Do not support the `long long` data type.

`--no_long_long` is supported only in ANSI and GNU C modes. It is not supported on any CPU with 64-bit registers, nor is it supported in INTEGRITY, u-velOSity, or embedded Linux platforms. In addition, certain target-specific header files may require the `long long` type to exist.

Alignment & Packing

This option category is contained within **C/C++ Compiler**.

These options control the alignment and packing of various data objects in memory. See also `#pragma pack (n)`, in “General Pragma Directives” on page 680.

Packing (Maximum Structure Alignment)

Controls the default maximum alignment of all objects of types `struct` and `class`. Permitted settings for this option are:

- **1-byte** (`-pack=1`)

- **2-byte (-pack=2)**
- **4-byte (-pack=4)**
- **8-byte (-pack=8)**

No `struct` or `class` or member of a `struct` or `class` will have an alignment greater than the specified value, unless the setting is overridden by the `#pragma pack` directive (see “General Pragma Directives” on page 680).

C/C++ Data Allocation

This option category is contained within **C/C++ Compiler**.

These options control the treatment of data in C and C++.

Allocation of Uninitialized Global Variables

Controls the allocation of uninitialized global variables. Permitted settings for this option are:

- **Treat as Common Data (--commons)** — [default] Treats a declaration at the outermost level of a C file such as `int foo;` as *common* data. Common data can be defined multiple times, and all declarations will be merged by the linker. For example, multiple files may have the declaration `int foo;` and the linker will merge them all into the same data entity.
- **Treat as Unique Definitions (--no_commons)** — Allocates uninitialized global variables to a section and initializes them to zero at program startup. This may improve optimizations by giving the compiler optimizer more information about the location of the variable.

However, if your code contains multiple declarations such as `int foo;` (which would cause the linker to find multiple definition spots for `foo`), this setting may also cause `multiply defined symbol` linker errors. In particular, you may not use a definition such as `int foo;` in a header file if it will be included in more than one source file. You can avoid these linker errors by using the `extern` keyword for declarations and having a single definition point. For example:

```
int foo;          /* used in just one place, or */  
  
int foo=0;        /* used in just one place */  
  
extern int foo; /* used in all other places */
```

Uniquely Allocate All Strings

This option is deprecated and may be removed in future versions of MULTI.

Controls the creation of a separate space for all strings, even those which are equivalent. Permitted settings for this option are:

- **On** (**--unique_strings**) — Create a separate space for all strings.
- **Off** (**--no_unique_strings**) — [default] Do not create separate spaces for each instance of an equivalent string.

Retain Symbols for Unused Statics

Controls the retention of symbols for variable and routines that are declared `static`, but are not used. Permitted settings for this option are:

- **On** (**--keep_static_symbols**) — Retain unused static variables and routines.
- **Off** (**--no_keep_static_symbols**) — [default] Discard unused static variables and routines.

Special Tokens

This option category is contained within **C/C++ Compiler**.

These options control the treatment of special tokens.

Alternative Tokens

Controls support for digraphs in C and C++. In C++, this option also controls whether operator keywords `and`, `or`, `bitand`, `bitor`, `and_eq`, `or_eq`, etc, are recognized. In C, the same tokens are provided as `#define` directives in **iso646.h**. Permitted settings for this option are:

- **On (--alternative_tokens)** — [default] Enables you to write C and C++ without using punctuation that might not be available on some international keyboards.
- **Off (--no_alternative_tokens)**

Host & Target Character Encoding

Controls support for EUC or Shift-JIS Kanji characters. Permitted settings for this option are:

- **EUC on Host and Target (-kanji=euc)** — [default for Solaris hosts] Specifies Kanji character interpretation using the EUC format on the host and target.
- **Shift-JIS on Host and Target (-kanji=shiftjis)** — [default for non-Solaris hosts] Specifies Kanji character interpretation using the Shift-JIS format on the host and target.
- **EUC on Host and Shift-JIS on Target (-kanji=euc/shiftjis)** — Specifies Kanji character interpretation using the EUC format on the host and the Shift-JIS format on the target.
- **UTF-8 on Host and Target (-kanji=utf8)** — Specifies UTF-8 as the multi-byte character encoding and UCS-2 as the wide-character encoding.
- **8-bit Codepage on Host and Target (-kanji=none)** — Use 8-bit code pages (for example, ISO 8859-1). In this mode there are no multibyte characters; the mapping from bytes to characters is one-to-one.
- **Shift-JIS on Host and UTF-8 on Target (-kanji=shiftjis/utf8)** — Specifies Kanji character interpretation using the Shift-JIS format on the host, UTF-8 as the target multi-byte character encoding, and UCS-2 as the target wide character encoding.
- **Shift-JIS on Host and UTF-16 on Target (-kanji=shiftjis/utf16)** — Specifies Kanji character interpretation using the Shift-JIS format on the host, Shift-JIS as the target multi-byte character encoding, and UCS-2 as the target wide character encoding.
- **Off (-kanji=none)** — Disables Kanji support.

C++

This option category is contained within **C/C++ Compiler**.

These options control aspects of compilation that relate to C++.

Namespaces

This option category is contained within **C/C++ Compiler→C++**.

These options control the treatment of namespaces.

Namespace Support

Controls support for namespaces. Permitted settings for this option are:

- **On** (**--namespaces**) — [default for C++ and EEC++]
- **Off** (**--no_namespaces**) — [default for EC++]

Implicit Use of 'std' Namespace

Controls the implicit use of the `std` namespace when standard header files are included. Permitted settings for this option are:

- **On** (**--using_std**)
- **Off** (**--no_using_std**) — [default]

Keyword Support

This option category is contained within **C/C++ Compiler→C++**.

These options control the treatment of various C++ keywords.

Bool Type Support

Controls support for the `bool` type. Permitted settings for this option are:

- **On** (**--bool**) — [default] Also defines the preprocessor symbol `_BOOL`, allowing code to determine when a `typedef` statement should be used to define the `bool` type.
- **Off** (**--no_bool**)

restrict Keyword Support

Controls support for the `restrict` keyword. Permitted settings for this option are:

- **On** (`--restrict`)
- **Off** (`--no_restrict`) — [default]

This is a C99 feature that can be used to indicate that a pointer is not aliased.

Support `__noinline` Keyword

Controls support for the `__noinline` keyword. This option is automatically enabled when you set the **Optimization**→**Optimization Strategy** option to **Optimize for Size (-Osize)**. Permitted settings for this option are:

- **On** (`--enable_noinline`)
- **Off** (`--disable_noinline`) — [default] The compiler ignores the `__noinline` keyword.

For more information, see “Additional C++ Inlining Information” on page 328.

Instantiate Extern Inline

Controls instantiation of `extern inline` functions and inline member functions of classes that have external linkage. Permitted settings for this option are:

- **On** (`--instantiate_extern_inline`) — Produces exactly one out-of-line copy in the program for each `inline` function that requires one. The prelinker ensures that there is one copy, unless you use the `--link_once_templates` option, in which case this is enforced with link-once sections (see “**Link-Once Template Instantiation**” on page 235). This option might increase compile time.
- **Off** (`--no_instantiate_extern_inline`) — [default] Produces an out-of-line copy of each `extern inline` function and each inline member function of a class with external linkage in each module that requires a copy. These out-of-line copies share static data. As a result:
 - If your code compares pointers to such functions from different modules, they will not be equal.

- Your code size may be larger.

For more information about inlining functions in C++, see “Manual Inlining” on page 318.

New-Style Cast Support

Controls the use of new-style casts (like `static_cast< >` and `reinterpret_cast < >`) with Embedded C++. Permitted settings for this option are:

- **On** (`--new_style_casts`)
- **Off** (`--no_new_style_casts`) — [default]

Using this option enables support for templates, which are usually not supported in Embedded C++.

Constructors/Destructors

This option category is contained within **C/C++ Compiler→C++**.

These options control the treatment of C++ constructors and destructors.

Support for Constructors/Destructors

Controls the insertion of code at the beginning of `main()` to call `_main()` which is responsible for invoking constructors on global objects that require them, and for invoking `atexit()` to cause the appropriate destructors to be invoked when the program finishes. Permitted settings for this option are:

- **On** (`--enable_ctors_dtors`) — [default]
- **Off** (`--disable_ctors_dtors`)

This option only controls the insertion of the call to `_main()` (and hence the inclusion of `_main()` from the C++ library) and does not affect the generation of code and data related to invoking constructors for global objects.

Placement of Class Constructor Call to New

Permitted settings for this option are:

- **Outside** (`--new_outside_of_constructor`) — Moves the call to `new` outside of a class constructor and inlines it inside the caller's code. Using this setting generates faster code in certain cases in which dynamic allocation is not needed (for instance, when the class object is allocated on the stack). However, using it might also result in larger code.
- **Inside** (`--new_inside_of_constructor`) — [default] Places the call to `new` inside the constructor, adding several instructions to the generated code.

RTTI Support

This option category is contained within **C/C++ Compiler**→**C++**.

These options control the treatment of *Run-Time Type Information* (RTTI).

Run-Time Type Information Support

Controls support for Run-Time Type Information (RTTI) features `dynamic_cast` and `typeid`. Permitted settings for this option are:

- **On** (`--rtti`) — [default for C++]
- **Off** (`--no_rtti`) — [default for EC++ and EEC++]

Treatment of RTTI as const

Controls the treatment of RTTI variables. Permitted settings for this option are:

- **On** (`--readonly_typeinfo`) — [default] Forces RTTI variables to be treated as `const` in the compiler. In most cases, `const` variables end up in the read-only data.
- **Off** (`--no_readonly_typeinfo`)

Virtual Tables

This option category is contained within **C/C++ Compiler**→**C++**.

These options control the treatment of virtual tables.

Virtual Function Definition

Controls the definition of virtual function tables in instances when the compiler heuristic provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline, non-pure virtual function of the class. For classes that contain no such function, the default is to define the virtual function table (but to define it as a local static entity). Permitted settings for this option are:

- **Force** (**--force_vtbl**) — Forces the definition of virtual function tables for such classes, but does not force the definitions to be local.
- **Standard** (**--standard_vtbl**) — [default] Defines the virtual function table as a local static entity.
- **Suppress** (**--suppress_vtbl**) — Suppresses the definition of virtual function tables for such classes.

Treatment of Virtual Tables as 'const'

Controls whether virtual tables are treated as `const`. Permitted settings for this option are:

- **On** (**--readonly_virtual_tables**) — [default] Forces virtual function tables to be treated as `const` in the compiler. In most cases, `const` tables end up in the read-only data.
- **Off** (**--no_readonly_virtual_tables**)

Templates

This option category is contained within **C/C++ Compiler**→**C++**.

These options control C++ template instantiation. For a full description of this feature, see “Template Instantiation” on page 634.

Link-Once Template Instantiation

Controls the link-once method of non-export template instantiation. Permitted settings for this option are:

- **On** (`--link_once_templates`)
- **Off** (`--no_link_once_templates`) — [default]

If `--one_instantiation_per_object` is used along with `--link_once_templates`, the templates are instantiated into their own uniquely named object files instead of link-once sections. These object files will be rewritten by a new file of the same name each time a source module that uses the template is compiled, preventing multiply defined symbols at link time.

Guiding Declarations of Template Functions

Controls recognition of *guiding declarations* of template functions. Permitted settings for this option are:

- **On** (`--guiding_decls`)
- **Off** (`--no_guiding_decls`) — [default]

A guiding declaration is a function declaration that matches an instance of a function template but has no explicit definition (since its definition derives from the function template). For example:

```
template <class T> void f(T t) {  
    // ...  
}  
void f(int);
```

When regarded as a guiding declaration, `f(int)` is an instance of the template; otherwise it is an independent function for which a definition must be supplied. If this option is disabled and **C/C++ Compiler→C++→Templates→Support for Old-Style Specializations** remains enabled, a specialization of a non-member template function is not recognized. It is treated as a definition of an independent function.

Implicit Source File Inclusion

Controls implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. For more information, see “Implicit Inclusion” on page 640. Permitted settings for this option are:

- **On** (`--implicit_include`)
- **Off** (`--no_implicit_include`) — [default]

Support for Implicit Typenames

Controls the implicit determination, from context, of whether a template parameter-dependent name is a type or a nontype. Permitted settings for this option are:

- **On** (`--implicit_typename`)
- **Off** (`--no_implicit_typename`) — [default]

Support for Old-Style Specializations

Controls the treatment of old-style template specializations (that is, specializations that do not use the `template<>` syntax). Permitted settings for this option are:

- **On** (`--old_specializations`)
- **Off** (`--no_old_specializations`) — [default]

One Instantiation Per Object

Controls whether template instantiations are put in separate object files. Permitted settings for this option are:

- **On** (`--one_instantiation_per_object`) — Puts each instantiation of a template or `extern inline` function (see “**Instantiate Extern Inline**” on page 231) in a separate template instantiation object file, with a unique filename that corresponds to the name of the instantiated entity. The primary object file (the object file corresponding to the original source file) contains everything else in the compilation; that is, everything that is not an instantiation.

The **gbuild** options **-clean** and **-cleanfirst** and the **clean** menu items in the Project Manager do not delete template instantiation object files (though they do delete primary object files).

- **Hybrid** (**--hybrid_one_instantiation_per_object**) — Puts each instantiation in the compilation in a separate object file, except where they are created by `#pragma instantiate`. Instances created by `#pragma instantiate` are added to the primary object file generated from the source file in which they appear.
- **Off** (**--no_one_instantiation_per_object**) — [default]

If you are creating a library, we recommend that you set this option to **On** or **Hybrid**. By putting each instantiation in a separate object file within the library, only those instantiations that are needed will be pulled in, thus reducing code size. One of these options must be enabled if two different libraries include some of the same instantiations, in order to avoid multiply defined symbols.

Template Instantiation Output Directory

When C/C++ **Compiler**→**C++**→**Templates**→**One Instantiation Per Object** is enabled (**--one_instantiation_per_object**), this option can be used to specify or create a *directory* into which the generated object files should be put. Multiple projects may not share an instantiation directory. The equivalent driver option is:

- **--instantiation_dir=directory**

The default instantiation directory is **./template_dir**.

If the directory setting used for **-instantiation_dir** is a relative path, that path is resolved relative to the Object File Output Directory (**-object_dir**). If **-object_dir** is not set, then the Instantiation Directory path is relative to the current working directory.

Prelink with Instantiations

Controls a mode of the Builder or driver that runs the C++ **prelink** utility to instantiate templates without running the linker or archiver. While this approach to template instantiation works with **clearmake**, we recommend that you use **--link_once_templates** instead, if it is available. Permitted settings for this option are:

- **On (--prelink_objects)** — Creates object files, but does not perform linking. The object files contain all template instantiations required, and so can be linked later without concern for template requirements.
- **Off (--no_prelink_objects)** — [default]

When using the driver, do not use this option in combination with any option that prevents the linker from being run, such as **-E**, **-P**, **-S**, or **-c**.

Dependent Name Processing

Controls the separate lookup of names in templates at the time the template is parsed and at the time it is instantiated. Permitted settings for this option are:

- **On (--dep_name)** — [default for the **GNU 4.3 (strict) C++** mode (**--g++ --gnu_version=40300_strict**)] Also implies **C/C++ Compiler→C++→Templates→Parse Templates in Generic Form (--parse_templates)**.
- **Off (--no_dep_name)** — [default for all other modes]

Parse Templates in Generic Form

Controls the parsing of nonclass templates in their generic form (i.e., even if they are not really instantiated). Permitted settings for this option are:

- **On (--parse_templates)**
- **Off (--no_parse_templates)** — [default]

This option is implied by **Dependent Name Processing** (see “**Dependent Name Processing**” on page 238).

Recognition of Exported Templates

Controls recognition of exported templates. Permitted settings for this option are:

- **On (--export)**
- **Off (--no_export)** — [default]

This option enables **C/C++ Compiler→C++→Templates→Dependent Name Processing** (**--dep_name**) (see above). It cannot be used with **C/C++ Compiler→C++→Templates→Implicit Source File Inclusion** (**--implicit_include**).

No levels of interprocedural optimizations are supported with export templates. If any level is selected it will be disabled.

Prelink File to Create Template Instances

Specifies an external object file or executable to notify the prelinker of existing template instances. The equivalent driver option is:

- **-prelink_against** *file*

Auto-Instantiation of Templates

Controls running the prelinker and output of template information and instantiation request files.

- **On** (**--auto_instantiation**) — [default]
- **Off** (**--no_auto_instantiation**) —

Disabling this option may result in faster builds and should be used in combination with **--link_once_templates**. Objects built with **--export**, **--one_instantiation_per_object**, or **--hybrid_one_instantiation_per_object** cannot be linked with **--no_auto_instantiation**.

Deferral of Function Template Parsing

Controls whether function templates are parsed immediately upon their definition, or delayed until there is an instantiation of the template. Permitted settings for this option are:

- **On** (**--defer_parse_function_templates**) — Parsing of function templates is delayed until there is an instantiation of the template. This behavior may allow syntax errors in templates that are not instantiated to go unnoticed by the compiler. This option is enabled by default in the **--g++** dialect with **--gnu_version=40300_strict**. (It is moot unless **C/C++ Compiler→C++→Templates→Parse Templates in Generic Form**)

(**--parse_templates**) is enabled, and that is not enabled by default for other emulated GNU versions.)

- **Off (--no_defer_parse_function_templates)** —Parsing of function templates happens immediately upon their definition. This option is enabled by default in all other dialects.

Assembler Options

This is a top-level option category.

These options control the general operation of the assembler.

For additional, more specialized options, see “Assembler Options” on page 309.

Source Listing Generation

Controls the generation of a source listing. Permitted settings for this option are:

- **Generate Default Listing (-list)** — Creates a listing by using the name of the object file with the **.lst** extension.
- **Generate User-Specified Listing (-list=*filename*)** — Creates a listing with the specified *filename*.
- **Suppress Listing (-no_list)** — [default]

Source Listing Generation Output Directory

Outputs assembly listing files in the specified directory. By default, listing files are output in the same directory as the object file. The equivalent driver option is:

- **-list_dir=*directory***

Preprocess Assembly Files

Controls whether assembly files with standard extensions such as **.s** and **.asm** are preprocessed. Permitted settings for this option are:

- **On (-preprocess_assembly_files)**
- **Off (-no_preprocess_assembly_files)** — [default] Only assembly files with special extensions are preprocessed.

Preprocess Special Assembly Files

Controls whether assembly files with a **.ppc** extension are preprocessed. Permitted settings for this option are:

- **On** (**-preprocess_special_assembly_files**) — [default]
- **Off** (**-no_preprocess_special_assembly_files**) — Files with a **.ppc** extension are not preprocessed.

Interleaved Source and Assembly

Controls the interleaving of your original source code with the generated assembly code. Normally used with the **Assembler→Source Listing Generation** option. Permitted settings for this option are:

- **On** (**-passsource**)
- **Off** (**-nopasssource**) — [default]

Not every line of the original source code will appear in the output.

Additional Assembler Options

Passes the specified assembler options to the **asppc** assembler command line. The equivalent driver option is:

- **-asm=options** — To pass multiple options, separate the options by spaces and enclose the whole string in quotes, or specify **-asm=options** multiple times.

For example:

```
-asm="-nogen -ref -w"
```

or:

```
-asm=-nogen -asm=-ref -asm=-w
```

For a full list of assembler options which may be passed in this manner, see Chapter 9, “The asppc Assembler”.

Assembler Command File

Passes the options specified in *file* directly to the assembler. The equivalent driver option is:

- **-asmcmd=***file*

The file must contain only one assembler option per line. For example:

```
-nogen  
-ref  
-w
```

For a full list of assembler options which may be passed in this manner, see Chapter 9, “The asppc Assembler”.

Support for C Type Information in Assembly

Controls assembler support for the `.inspect` and `directive`, and the `offsetof()` and `sizeof()` operators. These allow you to parse C header files to get type information, and use it to define objects in assembly. For more information, see Chapter 10, “Assembler Directives”, and “C Type Information Operators” on page 403. Permitted settings for this option are:

- **On (-asm3g)** — [default]
- **Off (-noasm3g)**

Linker Options

This is a top-level option category.

These options control the general operation of the linker.

For additional, more specialized options, see “Linker Options” on page 309.

Output File Type

Controls the form of linker output. Permitted settings for this option are:

- **Executable / Located Program (-locatedprogram)** — [default] Generates an executable program.
- **Relocatable Object File (-relobj)** — Generates a relocatable object file, suitable for passing as input to another run of the linker. Implies **-nostdlib**, meaning the linker does not link in any startup files or libraries, and **-undefined**, meaning the linker does not allocate common variables or give errors for undefined symbols, as **Relocatable Program** does.

If you are generating debug information for a relocatable object file with **-G** and the machine that will be performing the subsequent link does not have access to the original **.o** or **.dbo** files, pass the **-search_for_dba** option. This option creates a **.dba** file that you can distribute with the relocatable object file to provide debug information to MULTI. When performing subsequent links with the relocatable object file, continue to pass the **-search_for_dba** option. This option passes **-r** to the linker.

- **Relocatable Program (-relprog)** — Retains relocation information in the output file. The resulting file is suitable for execution. Some of the final link steps, including but not limited to C++ constructors and special symbols, are not guaranteed to have relocations, and thus might not be valid if the output file is loaded at a different address. This option passes **-a** to the linker.
- **Relocatable program Cafe OS (-relprog_cafe)** — Produce an executable suitable for Cafe OS. Retains relocation information in the output file.

Generate Additional Output

Creates the specified output type in addition to the project executable. Permitted settings for this option are:

- **Memory Image File (-memory)** — Generates output file with `.mem` extension containing the output of the image as translated by the **gmemfile** utility program. For more information about **gmemfile**, see “The gmemfile Utility Program” on page 524.
- **S-Record File (-srec)** — Generates output file with `.run` extension containing the output of the image as translated by the **gsrec** utility program. For more information about **gsrec**, see “The gsrec Utility Program” on page 544.
- **HEX386 File (-hex)** — Generates output file in HEX386 format with `.run` extension containing the output of the image as translated by the **gsrec** utility program with **-hex386** passed to it. For more information about **gsrec**, see “The gsrec Utility Program” on page 544.
- **None (--no_additional_output)** — [default] You can use this option in a MULTI Project (**.gpj**) file, but there is no equivalent driver option.

Executable Stripping

Controls *stripping* the executable at the conclusion of linking. Permitted settings for this option are:

- **On (-strip)** — Stripping involves removing line number, symbol table, and debugging information to reduce the file size of the executable.
- **Off (-nostrip)** — [default]

For information about stripping executables after link-time, see “The gstrip Utility Program” on page 554.

Start Address Symbol

Specifies the *symbol* whose address is used as the program entry point or start address. Permitted settings for this option are:

- **Start Address (-e *symbol*)** — The default symbol is `_start`.
- **No Entry Symbol (-noentry)**

Append Comment Section with Link-Time Information

Controls the printing of the tools version number and a timestamp to the output of the linker. Similar to using `#pragma ident` (see “General Pragma Directives” on page 680). Permitted settings for this option are:

- **On** (`-Qy`)
- **Off** (`-Qn`) — [default]

Preprocess Linker Directives Files

Controls the level of preprocessing performed on linker directives files. Permitted settings for this option are:

- **Full** (`--preprocess_linker_directive_full`) — The C preprocessor preprocesses linker directives files.
- **Partial** (`--preprocess_linker_directive`) — The C preprocessor resolves preprocessing directives such as `#include`, `#if`, and `#ifdef`. The preprocessor expands macros in preprocessing directives, but not in the rest of the file. This option is deprecated and may be removed in future versions of **MULTI**.
- **Off** (`--no_preprocess_linker_directive`) — [default]

Linker Warnings

Controls the display of linker warnings. Permitted settings for this option are:

- **Display** (`-linker_warnings`) — [default]
- **Suppress** (`-no_linker_warnings`)

Raw Import Files

Imports data from arbitrary files into a section named `.raw`. The equivalent driver option is:

- **-rawimport**

If your section map does not specify a section named `.raw`, the linker issues a warning. To silence the warning, add a section named `.raw` to your section map.

Alternatively, you can select a section for each imported file using section inclusion commands in your linker directives (**.ld**) file. For example, to import the file **rawdata.bin** and place the data in the section `.myrawdata`, use **-rawimport rawdata.bin** and add the following line to your section map:

```
.myrawdata : {rawdata.bin(.raw) }
```

By default, imported sections become read-only data sections. To make a section writable or executable, use the SHFLAGS section attribute. For more information, see “Using Section Attributes” on page 451 and “Defining a Section Map with the SECTIONS Directive” on page 442.

Linker Directive Files with Non-standard Extensions

Passes *file* with a non-standard extension to the linker as a linker directives file, instead of any default file. You can use this option multiple times.

The equivalent driver option is:

- **-T** *directives_file*

The following list describes how the compiler interprets **-T** depending on the name of *file*:

- *file* has a known linker directives extension (**.ld**) — While the compiler passes the file to the linker, **-T** is not necessary. In this case, pass just the filename to the driver.
- *file* has an extension commonly used for source files or object files (**.c**, **.obj**, **.a**, etc.) — The compiler issues the following warning and ignores **-T**:

```
Warning: Option "-T" ignored due to invalid suffix of argument
file. Use .ld as suffix
```

- *file* begins with a hyphen (**-**) — The compiler issues the following warning and ignores **-T**:

```
Warning: Option "-T" ignored due to invalid argument -file.
expected directory or filename, without leading -
```

Additional Linker Options (beginning of link line)

Passes the specified linker *options* to the **elxr** linker command line at the very beginning of the line, before any **.ld** files, and before any other options. The equivalent driver option is:

- **-lnk0=options** — To pass multiple options, separate the options by spaces and enclose the entire string in quotes, or specify **-lnk0=options** multiple times.

For example:

```
-lnk0="-multiple -undefined"
```

or:

```
-lnk0=-multiple -lnk0=-undefined
```

For more information about linker-specific options, see “Linker-specific Options” on page 434.

For additional information about passing linker options directly, see “**Additional Linker Options (before start file)**” on page 248.

Additional Linker Options (before start file)

Passes the specified linker *options* to the linker command line after any **.ld** files and after most default options, but before the start files. The equivalent driver option is:

- **-lnk=options** — To pass multiple options, separate the options by spaces and enclose the whole string in quotes, or specify **-lnk=options** multiple times.

For example:

```
-lnk="-multiple -undefined"
```

or:

```
-lnk=-multiple -lnk=-undefined
```

There are three options for passing options through to the linker command line. Each of the three options places its argument on the linker command line in a different location as described in the following list:

- **-lnk0=** should be used for options that must appear at the very beginning of the linker command line, before the linker directives files, and before any options that are passed by the driver.
- **-lnk=** causes the options to appear after many of the drivers options, but before any object files. Options that enable or disable a feature have a higher precedence if passed with **-lnk=** than with **-lnk0=**.
- **-Wl** should be used in the rare case that an option must appear in between object files on the linker command line, or if an option must appear at the end of the linker command line. The position of the **-Wl** option on the driver command line determines its position on the linker command line, relative to all source and object files.

For more information about linker-specific options, see “Linker-specific Options” on page 434.

Additional Linker Options (among object files)

Passes the specified linker options to the linker in approximately the position that it appears on the driver command line. The equivalent driver option is:

- **-Wl,option[,option]...**

For more information about linker-specific options, see “Linker-specific Options” on page 434.

For additional information about passing linker options directly, see “**Additional Linker Options (before start file)**” on page 248.

Linker Command File

Passes the options specified in *file* directly to the linker. The equivalent driver option is:

- **-lnkcmd=***file*

The file must contain only one linker option per line. For example:

```
-nogen  
-ref  
-w
```

For more information about linker-specific options, see “Linker-specific Options” on page 434.

Linker Optimizations

This option category is contained within **Linker**.

These options control individual optimizations that can be implemented at link-time. You can enable or disable all of them with the **Optimization→Linker Optimizations** option (see “Optimization Options” on page 176).

Deletion of Unused Functions

Controls the removal from the executable of functions that are unused and unreferenced. Permitted settings for this option are:

- **On (-delete)**
- **Off (-no_delete)** — [default]

Certain options that create references to functions may inhibit this option. For more information, see “Deleting Unused Functions” on page 467.

Deletion of Unused Data

Controls the removal from the executable of data that are unused and unreferenced. Permitted settings for this option are:

- **On (-data_delete)** — [default]
- **Off (-no_data_delete)**

Note that this option has no effect unless the **-delete** option is enabled.

Data can only be deleted from object files generated by the GHS compiler

Deletion of Unused C++ Virtual Functions

Enables the deletion of C++ virtual functions. This options requires that all object files in the program be compiled with Green Hills compilers. Permitted settings for this option are:

- **On** (**-uvfd**)
- **Off** (**-no_uvfd**)

Code Factoring

Controls the code factoring optimization, which reduces code size by merging redundant sequences of object code at link-time. Permitted settings for this option are:

- **On** (**-codefactor**)
- **Off** (**-no_codefactor**) — [default]

For more information see “Code Factoring” on page 469.

Link-Time Constant Propagation

Controls the link-time constant propagation optimization. Permitted settings for this option are:

- **On** (**--link_constprop**) — Instructs the linker to optimize the program further when new constants are resolved at link time, for example, when weak symbols are defined/left undefined during the final link (i.e. the constant 0 will be propagated for weak undefined symbols).
- **Off** (**--no_link_constprop**) — [default]

Link-Time Trivial Function Inlining

Inlines calls to trivial functions (return instructions) without deleting the callee function. Permitted settings for this option are:

- **On** (**-inline_trivial**)
- **Off** (**-no_inline_trivial**) — [default]

This option should be enabled with **-Olink** (see “Linker Optimizations” in “Optimization Options” on page 176).

Link-Time Ignore Debug References

Controls whether or not the linker ignores relocations from DWARF debug sections when you are using the **-delete** option. Permitted settings for this option are:

- **On (-ignore_debug_references)** — Ignores relocations from DWARF debug sections when using **-delete**. DWARF debug information will contain references to deleted functions that may break some third-party debuggers.
- **Off (-no_ignore_debug_references)** — Does not ignore relocations from DWARF debug sections when using **-delete**. DWARF debugging information generated by **-dual_debug** greatly reduces the effectiveness of **-delete**.

Start and End Files

This option category is contained within **Linker**.

These options control the use of start and end files.

Start Files

Controls the start files to be linked into the executable. The default is to use the Green Hills start files. Permitted settings for this option are:

- **Use Specified Start Files (-startfiles=file[,file...])**
- **Do Not Use Start Files (-nostartfiles)**

End Files

Specifies end files to be linked into the executable. End files are the last object files passed on the command line to the linker. The equivalent driver option is:

- **-endfile=file[,file...]**

Start File Directory

Specifies the directory that contains start files, end files, and the default linker directives file. This option has no effect on the location of the default

linker directives file if you also pass **-directive_dir** or if you are building an INTEGRITY project (unless it is a kernel). Use the **-startfiles** and **-endfile** options to override the default selection of start and end files, respectively. The equivalent driver option is:

- **-startfile_dir=directory**

Symbols

This option category is contained within **Linker**.

These options control the linker's treatment of symbols.

Multiply-Defined Symbols

Controls the treatment of multiply-defined symbols. Permitted settings for this option are:

- **Errors (-no_multiple)** — [default]
- **Silent (-multiple)**

Undefined Symbols

Controls the treatment of undefined symbol references. Permitted settings for this option are:

- **Errors (-no_undefined)** — [default]
- **Silent (-undefined)** — The linker gives each undefined symbol an address of zero, or a zero offset if you are using PID. If the symbol would normally be assigned to a special data area, it is given an address in the appropriate special data area section.

Define Linker Constant

Sets a default value that can be used in section and memory maps. This option overrides any default value specified in a linker directives file. The equivalent driver option is:

- **-Cname=value**

Force Undefined Symbol

Forces an undefined symbol reference for *symbol*, as if there had been some use of the symbol in one of the modules to be linked. This may cause modules to be linked in from libraries that would not otherwise be included. The equivalent driver option is:

- **-u** *symbol*

Force All Symbols From Library

Pulls in every exported symbol from the specified library, instead of pulling in only those symbols that are required. The equivalent driver option is:

- **-extractall=library**

Where the *library* parameter must exactly match the way that the library is specified on the command line. For example, use `-ltest` for the *library* parameter instead of `libtest.a`.

To pull in all the symbols from two or more libraries, pass multiple **-extractall** options, or pass one **-extractall** option with a comma-separated list of library names. For example, the following are equivalent:

```
-extractall=-lfoo, -lbar  
-extractall=-lfoo -extractall=-lbar
```

Import Symbols

Makes available symbol names and addresses from a separately linked, fully located file during linking. The linker does not include the contents of the file in the output; it only imports those symbol addresses necessary for the current link. This option is useful when one linker image must refer to symbols that are located in another separately linked image. The equivalent driver option is:

- **-A** *file*

Where *file* must be a file linked with the **-locatedprogram** option.

Record Pathname (Not linked)

Records strings in a symbol table. The equivalent driver option is:

- **-UA** *string*

Linker Output Analysis

This option category is contained within **Linker**.

These options control various forms of linker output.

Map File Generation

Controls the generation of a map file. Permitted settings for this option are:

- **Generate Default Map File (-map)** — [default] Creates a map file with the name of the object file plus a **.map** extension.
- **Generate User-Specified Map File (-map=*filename*)** — Creates a map file with the specified *filename*.
- **Suppress Map (-nomap)**

Map File Retention

Controls the retention of the map file in the event of a link error. Permitted settings for this option are:

- **On (-keepmap)**
- **Off (-nokeepmap)** — [default]

Map File Page Length

Specifies the length, *n*, of a map file page. The equivalent driver option is:

- **-maplines=*n***

Map File Sorting

Controls the method for ordering the map file. Permitted settings for this option are:

- **Alphabetic (-Ma)** — [default]
- **Numeric (-Mn)**

Map File Cross-Referencing

Controls the generating of cross-reference information in the map file. Permitted settings for this option are:

- **On (-Mx)**

Display Unused Functions in Map File

Marks unreferenced symbols in the map file.

Permitted settings for this option are:

- **On (-Mu)** — Adds an extra column to the map file's table of symbols that contains D for symbols which are unreferenced and therefore able to be deleted. This includes symbols that have already been deleted by the **-delete** option (if passed). This column contains U for any other symbols that remain unresolved, including undefined weak symbols. It is possible that some unreferenced symbols are not marked with D, either because they overlap with reachable symbols or because they are referenced from profiling or debugging information. For more information, see “Deleting Unused Functions” on page 467.

Display Local Symbols in Map File

Permitted settings for this option are:

- **On (-Ml)** — Adds list of locals to the linker-generated map file. This list does not contain zero-sized symbols or compiler-generated symbols.

Output File Size Analysis

Controls use of the **gsize** utility to determine the size of the output executable. **gsize** creates a file called *executable_name.siz*, which contains a list of the sizes of the ROM sections of the output executable. Permitted settings for this option are:

- **On (-gsize)**
- **Off (-no_gsize)** — [default]

For more information, see “The gsize Utility Program” on page 542.

Call Graph Generation

Controls the generation of a call graph. Permitted settings for this option are:

- **Generate Default Call Graph (-callgraph)** — Creates a call graph with the name of the object file plus a **.graph** extension.
- **Generate User-Specified Call Graph (-callgraph=filename)** — Creates a call graph with the specified *filename*.
- **Suppress Call Graph (-no_callgraph)** — [default] You can use this option in a MULTI Project (**.gpj**) file, but there is no equivalent driver option.

Link-Time Checking

This option category is contained within **Linker**.

These options control various forms of linker checking.

Section Overlap Checking

Controls the treatment of overlapping sections. Permitted settings for this option are:

- **Errors on Any Overlap (-strict_overlap_check)**
- **Errors on Non-Zero Overlap (-nooverlap)** — [default]
- **Warnings (-overlap_warn)**
- **Silent (-overlap)**

Checksum

Controls the creation of section checksums. Permitted settings for this option are:

- **On (-checksum)** — Appends a 4-byte checksum to the end of every initialized program section using a standard 32-bit CRC algorithm.
- **Off (-nochecksum)** — [default]

The algorithm is a standard 32-bit CRC using the polynomial 0x10211021 (by default), or 0x04C11DB7 (for INTEGRITY). For information about how to use these checksums, see “Verifying Program Integrity” on page 150. Sample source code for verifying checksums is included in **src/libstartup/cksum.c** of your installation.

Compiler Diagnostics Options

This is a top-level option category.

These options control the type, form, and quantity of diagnostic messages you may receive during building.

Warnings

Controls the display of *warnings* for most Green Hills tools. Permitted settings for this option are:

- **Display** (`--warnings`) — [default]
- **Suppress** (`-w`)

Remarks

Controls the display of *remarks* for most Green Hills tools. Permitted settings for this option are:

- **Display** (`--remarks`)
- **Suppress** (`--no_remarks`) — [default]

Maximum Number of Errors to Display

Limits to n the number of error messages the compiler will print before quitting. The default is 100 and the minimum is 2.

The equivalent driver option is:

- **`-errmax=n`**

Some errors are catastrophic, and the compiler stops the compilation process after encountering such an error, regardless of whether the error limit has been reached.

Redirect Error Output to File

Specifies a *file* to which all error output is redirected. This option is not supported when parallel build mode is enabled (**-parallel**). The equivalent driver option is:

- **-stderr=***file*

Quit Building if Warnings are Generated

Controls whether the build will terminate with an error if any warnings are generated. This option provides the ability to treat all warnings as errors. Permitted settings for this option are:

- **On** (**--quit_after_warnings**)
- **Off** (**--no_quit_after_warnings**) — [default]

Display Version Information

Instructs the compiler and other tools to print its copyright banner and version number. Permitted settings for this option are:

- **On** (**-V**)
- **Off** (**--no_version**) — [default]

Varying Message Format

This option category is contained within **Compiler Diagnostics**.

These options control various aspects of the display of diagnostic messages.

Brief Error Message Mode

Controls a mode in which a shorter form of the diagnostic output is used. When enabled, the original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line. Permitted settings for this option are:

- **On** (**--brief_diagnostics**)

- **Off** (`--no_brief_diagnostics`) — [default]

Line Wrap Messages

Controls whether diagnostic messages are wrapped when they are too long to fit on a single line. Permitted settings for this option are:

- **On** (`--wrap_diagnostics`) — [default]
- **Off** (`--no_wrap_diagnostics`)

Display Error Message Paths

Controls whether full pathnames are given for files mentioned in diagnostic messages. Permitted settings for this option are:

- **On** (`-error_basename`) — Filenames in diagnostic messages include only the name of the file. No path information is provided.
- **Off** (`-no_error_basename`) — [default] — Filenames in diagnostic messages include the path information that was passed to the driver.

C/C++ Messages

This option category is contained within **Compiler Diagnostics**.

These options control various diagnostic messages specific to the C and C++ languages.

Functions Without Prototypes

Controls the treatment of functions that are referenced or called when no prototype has been provided. Permitted settings for this option are:

- **Errors** (`--prototype_errors`)
- **Warnings** (`--prototype_warnings`) — [default]
- **Silent** (`--prototype_silent`)

Asm Statements

Controls the treatment of `asm` statements. Permitted settings for this option are:

- **Errors** (`--asm_errors`)
- **Warnings** (`--asm_warnings`) — [default for strict ANSI C and strict ANSI C++]
- **Silent** (`--asm_silent`) — [default for all other C and C++ modes]

In strict ANSI C, the spelling `asm` is not part of the C language. It is unknown to the compiler and may be used as the name of a variable or function. To write an `asm` macro in strict ANSI C or strict ISO C99, you must use `__asm`. In non-strict ANSI C and ISO C99, and in C++, the two spellings (`asm` and `__asm`) are identical.

Any correct use of `__asm` may still give a diagnostic indicating that `__asm` is non-standard.

Unknown Pragma Directives

Controls the treatment of `#pragma` directives that are not recognized by the compiler. Permitted settings for this option are:

- **Errors** (`--unknown_pragma_errors`)
- **Warnings** (`--unknown_pragma_warnings`) — [default]
- **Silent** (`--unknown_pragma_silent`)

Incorrect Pragma Directives

Controls the treatment of valid `#pragma` directives that use the wrong syntax. Permitted settings for this option are:

- **Errors** (`--incorrect_pragma_errors`)
- **Warnings** (`--incorrect_pragma_warnings`) — [default]
- **Silent** (`--incorrect_pragma_silent`)

printf & scanf Argument Type Checking

Controls argument type checking against the format string in calls to `printf()` and `scanf()`. The checking is only performed if the format string is a constant. Permitted settings for this option are:

- **On (-Wformat)** — The compiler performs argument type checking against `printf()` and `scanf()` format strings. This option also defines the `__CHECK_PRINTF__` macro.
- **Off (-Wno-format)** — [default] The compiler does not perform argument type checking against `printf()` or `scanf()` format strings.

This option has no effect if Green Hills header files are not used.

Implicit Int Return Type

Controls a warning that is issued if the return type of a function is not declared before it is called (thus causing the return type to be implicitly declared `int`). Permitted settings for this option are:

- **On (-Wimplicit-int)**
- **Off (-Wno-implicit-int)** — [default]

For broader prototype checking, see **Compiler Diagnostics**→**C/C++ Messages**→**Functions Without Prototypes**.

Shadow Declarations

Controls a warning that is issued if the declaration of a local variable *shadows* the declaration of a variable of the same name declared at the global scope, or at an outer scope. Permitted settings for this option are:

- **On (-Wshadow)**
- **Off (-Wno-shadow)** — [default]

Trigraphs

Controls a warning that is issued for any use of trigraphs. Permitted settings for this option are:

- **On** (-Wtrigraphs)
- **Off** (-Wno-trigraphs) — [default]

Undefined Preprocessor Symbols

Controls a warning that is issued for undefined symbols in preprocessor expressions. Such undefined symbols are always given the value of 0. Permitted settings for this option are:

- **On** (-Wundef)
- **Off** (-Wno-undef) — [default]

Varying Message Severity

This option category is contained within **Compiler Diagnostics→C/C++ Messages**.

These options allow you to increase or decrease the severity of any of the *discretionary* C and C++ compiler diagnostic messages. See “Display Error Message Numbers” for more information.

Values for the error number, *n*, can be obtained by enabling the **Compiler Diagnostics→C/C++ Messages→Varying Message Severity→Display Error Message Numbers (--display_error_number)** option (see below). The complete set of error numbers is also listed in the *MULTI: C and C++ Compiler Error Messages* book (not available in print).

Set Message to Error

Sets the specified diagnostics message to the level of *error*. The equivalent driver option is:

- **--diag_error** *n[,n2]...*

Set Message to Warning

Sets the specified diagnostics message to the level of *warning*. The equivalent driver option is:

- **--diag_warning** *n*[,*n2*]...

Set Message to Remark

Sets the specified diagnostics message to the level of *remark*. The equivalent driver option is:

- **--diag_remark** *n*[,*n2*]...

Set Message to Silent

Sets the specified diagnostics message to the level of *silent*. The equivalent driver option is:

- **--diag_suppress** *n*[,*n2*]...

Display Error Message Numbers

Controls the display of error message numbers in any diagnostic messages that are generated. Permitted settings for this option are:

- **On** (**--display_error_number**) — [default]
- **Off** (**--no_display_error_number**)

This option may be used to determine the error number to be used when overriding the severity of a diagnostic message.

The *diagnostic* number will be followed by **-D** if the diagnostic is discretionary, meaning that its severity may be modified. For example:

```
"hello.c", line 24: warning #549-D:  
variable "i" is used before its value is set
```

The displayed diagnostic is number 549 and is discretionary, so it may be suppressed or have its severity changed by the preceding options.

DoubleCheck (C/C++) Options

This is a top-level option category.

DoubleCheck Level

Each level performs a different degree of processing. The higher the level, the more bugs can be identified, and a longer processing time is required. Permitted settings for this option are:

- **None (-double_check.level=none)** — [default] DoubleCheck is disabled.
- **Low (-double_check.level=low)** — Looks for all types of bugs, but does not follow execution paths involving function calls. DoubleCheck runs quickly and should not greatly affect the time required to compile files.
- **Medium (-double_check.level=medium)** — Looks for all types of bugs, and follows execution paths through calls between functions in all source files. This extra processing requires more time than the low level.
- **High (-double_check.level=high)** — Extends the medium level by investigating even more interprocedural execution paths. This requires more processing time than the medium level.

The medium and high levels require two-pass compilation. The first pass summarizes the contents of source files. The second pass uses the summaries of all source files when compiling each file to identify potential intermodule paths.

For more information about DoubleCheck and these options, see Chapter 8, “The DoubleCheck Source Analysis Tool”.

DoubleCheck Report File

Sets the name of the report file DoubleCheck generates. The equivalent driver option is:

- **-double_check.report=**

For more information about **-double_check.report=**, see “Specifying a DoubleCheck Report File” on page 367.

DoubleCheck Config File

Specifies the configuration file containing function properties. The equivalent driver option is:

- **-double_check.config=**

For more information about **-double_check.config=**, see “Specifying Function Properties in a DoubleCheck Configuration File” on page 369.

DoubleCheck Errors to Ignore

Disables a particular DoubleCheck error message or warning message for a specific file or part of a build hierarchy. The equivalent driver option is:

- **-double_check.ignore=**

For more information about **-double_check.ignore=**, see “Ignoring Source Analysis Errors and Warnings” on page 375.

DoubleCheck Output that Stops Builds

Specifies whether DoubleCheck error or warning messages stop the build process. Permitted settings for this option are:

- **Off (-double_check.stop_build=off)** — [default] DoubleCheck error and warning messages do not stop the build.
- **Errors Only (-double_check.stop_build=errors)** — Only DoubleCheck error messages stop the build.
- **Warnings and Errors (-double_check.stop_build=warnings)** — Both DoubleCheck error and warning messages stop the build.

For more information about **-double_check.stop_build=**, see “Promoting Source Analysis Errors and Warnings” on page 374.

HTML Compiler Options

This is a top-level option category.

The HTML compiler is deprecated and may be removed in future versions of **MULTI**. The options in this section are provided for backwards compatibility only.

Alternate Source Name

This option is deprecated and provided for backwards compatibility only.

Renames an output file. The equivalent HTML compiler option is:

- **-o**

HTML File Compression

This option is deprecated and provided for backwards compatibility only.

Omits HTML file compression. Permitted settings for this option are:

- **No HTML Compression (-c)**
- **HTML Compression (--enableCompression)**

Alternate Compression Table

This option is deprecated and provided for backwards compatibility only.

Filename of alternate compression table. The equivalent HTML compiler option is:

- **-p**

Advanced Options

This is a top-level option category.

Target Options

This option category is contained within **Advanced**.

These options control advanced settings that affect compilation for your target. For more common options, see “Target Options” on page 154.

Target Processor

Specifies code generation for a particular target *processor*.

If you specify a processor, the driver selects the appropriate instruction set and a generic **.ld** file. For more information about these **.ld** files, see “Working with Linker Directives Files” on page 63. You can override the default **.ld** file by specifying an alternate linker directives file elsewhere on the command line. For a more detailed discussion on **.ld** files, see “Configuring the Linker with Linker Directives Files” on page 437.

The equivalent driver option is:

- **Generic CPU (-cpu=cpu)** — To obtain a list of supported processors from the command line, enter **-cpu=?** (or **-cpu=\?** in many shells).

Additional SDA Base Registers

Specifies up to ten additional SDA and ZDA base pointers, beginning at R14. The equivalent driver option is:

- **-additional_sda_reg=n**

For more information, see “Specifying Additional Registers for Special Data Areas” on page 145.

Generation of `fsel` Instructions

Controls the automatic generation of the `fsel` floating-point conditional move instruction. Permitted settings for this option are:

- **On** (`-use_fsel`) — This is the default setting for processors that support `fsel` when hardware floating-point support is selected.
- **Off** (`-no_use_fsel`) — This is the default setting for other processors, or when software floating-point support is selected.

Generation of `isel` Instructions

Controls the automatic generation of the `isel` integer conditional move instruction. Permitted settings for this option are:

- **On** (`-isel`) — [default for e500 and 440EP/440GX processors]
- **Off** (`-no_isel`) — [default for other processors]

Expansion of e500 `evmwhs__a_w` Intrinsics

Controls the automatic expansion of calls to intrinsics using the e500 `evmwhssfaaw`, `evmwhssiaaw`, `evmwhsmfaaw`, `evmwhsmiaaw`, `evmwhusiaaw`, `evmwhumiaaw`, `evmwhssfanw`, `evmwhssianw`, `evmwhsmfanw`, `evmwhsmianw`, `evmwhusianw`, or `evmwhumianw` instructions.

Permitted settings for this option are:

- **On** (`-e500_inst_fix`) — [default for e500 processors]
- **Off** (`-no_e500_inst_fix`) — [default for other processors]

Expansion of e500 `evmwhgs__a_` Intrinsics

Controls the automatic expansion of calls to intrinsics using the e500 `evmwhgssfaa`, `evmwhgsmfaa`, `evmwhgsmiaa`, `evmwhgumiaa`, `evmwhgssfan`, `evmwhgsmfan`, `evmwhgsmian`, or `evmwhgumian` instructions.

Permitted settings for this option are:

- **On** (**-e500_inst_fix2**) — [default for e500 processors]
- **Off** (**-no_e500_inst_fix2**) — [default for other processors]

Allow Use of SPE instructions

This option only applies to processors that support SPE, such as the PowerPC 8540.

Permitted settings for this option are:

- **On** (**-SPE**) — The compiler generates SPE and vector floating point instructions, and uses C and C++ SPE extensions. When using this option, MSR [SPE] must be enabled.
- **Off** (**-noSPE**) — The compiler does not generate SPE or vector floating point instructions, and does not use C and C++ SPE extensions. **-noSPE** also passes **-noSPE** to the assembler (see “PowerPC-Specific Assembler Options” on page 394). Use this option when MSR [SPE] is disabled.

This option does not affect the generation of single precision embedded scalar floating point (on e500v1 and e500v2 processors) or double precision embedded scalar floating point (on e500v2 processors). Because double precision embedded scalar floating point on e500v2 processors requires MSR [SPE] to be set, this option is not supported on e500v2 processors.

Early versions of the e500 documentation used SPE to refer to the union of SPE, single precision embedded vector floating point, and single precision embedded scalar floating point. The definition of SPE used here is consistent with current e500 documentation and refers only to the integer vector instructions.

The standard libraries for processors that support SPE have been built to minimize the use of SPE instructions. The only places in the library that require MSR [SPE] under the **-noSPE** option are the startup and reset routines that set MSR [SPE], the routines `setjmp` and `longjmp`, which save and restore the full 64-bit registers using the SPE instructions, and the routines `memset` and `memcpy`, which use SPE instructions. By avoiding, replacing, or rebuilding these library routines, it is possible for applications to completely avoid the use of MSR [SPE].

Always Use `lmw/stmw` in Interrupt Subroutine Prologue/Epilogue

Controls when the compiler uses the load and store multiple instructions `lmw` and `stmw` in interrupt subroutine prologues and epilogues. Permitted settings for this option are:

- **On** (`-interrupt_prologue_always_uses_stm`) — Forces the compiler to generate `lmw` and `stmw` instructions in interrupt subroutine prologues and epilogues whenever it must save more than one register. Turning this option on may affect program speed, because these instructions have more overhead than `lwz` and `stw`.
- **Off** (`-no_interrupt_prologue_always_uses_stm`) — [default] Instructs the compiler to make the optimal decision between using load and store multiple instructions or a sequence of loads or stores, based on the setting for **Optimization**→**Optimization Strategy**. If you are optimizing for size (`-Osize`), the compiler always uses `lmw` and `stmw`. If you are optimizing for speed (`-Ospeed`), the compiler sometimes uses sequences of `lwz` and `stw` because of the overhead involved in the other instructions.



Note For PowerPC 560xB, 560xP, 560xS, and 563xM processors, the compiler generates `lmvgprw` and `stmvgprw` instead of `lmw` and `stmw` to save volatile registers (`r0` and `r3-r11`) when they are the only registers that need to be saved.

Only explicit floating point and SIMD register use

Controls the implicit use by the compiler of floating point registers. When this option is on, the compiler will not generate code that uses floating point registers unless the user's code explicitly uses these types. When this option is off, the compiler is free to use these registers implicitly, for instance in structure copies. Permitted settings for this option are:

- **On** (`-only_explicit_reg_use`)
- **Off**

Function Prologues

Controls the manner in which the compiler generates each function prologue and epilogue. Permitted settings for this option are:

- **On (-inline_prologue)** — Forces the compiler to use inline code sequences. This option may adversely impact the size of the generated code, so it should only be used when necessary (for example, when the routines may not exist in memory yet).
- **Off (-no_inline_prologue)** — [default] This option selects the most efficient method, given the optimization settings and the registers that must be saved.

Support for Very Large Switch Statements

Controls the size of switch statement offset entries. Permitted settings for this option are:

- **On (-bigswitch)** — [default] Use a 32-bit offset. This option is set to **off** by default if **Optimization→Optimization Strategy** is set to **Optimize for Size (-Osize)**.
- **Off (-nobigswitch)** — Use a 16-bit offset, which is smaller and faster, but which will fail if a label is too far away.

Project Options

This option category is contained within **Advanced**.

These options control advanced project configuration settings. For more common options, see “Project Options” on page 172.

Binary Code Generation

Controls binary code generation. By default, the toolchain generates binary object files directly from source code rather than via the creation of assembly language files. Since the assembler is not invoked, binary code generation significantly decreases compile time.

Certain options, such as **-dual_debug**, are incompatible with binary code generation, and binary code generation will default to off if these are selected. Also, binary code generation is automatically disabled on individual files if the compiler determines that the file uses features (such as embedded assembly) that are incompatible with binary code generation.

This option does not control the output file format. To generate a binary output file, see the `-memory` driver option in “**Generate Additional Output**” on page 244.

Permitted settings for this option are:

- **On (-obj)** — Enables direct binary generation.
- **Off (-noobj)** — Disables direct binary generation.

Temporary Output Directory

Specifies a *directory* for writing temporary files to. This option is useful if your default temporary directory is on a small file system that might run out of disk space during compilations which involve inlining or template processing. The equivalent driver option is:

- **-tmp=directory**

(Linux/Solaris) By default, temporary files are written to the first of the following directories that exists:

- The directory specified by the environment variable `TMPDIR`.
- The directory specified by the constant value of variable `P_tmpdir` (usually `/var/tmp` on Solaris and `/tmp` on Linux), which is defined in the operating system header file (usually located in `/usr/include/stdio.h`).
- `/tmp`.

(Windows) By default, temporary files are written to the first of the following directories that exists:

- The directory specified by the environment variable `TMP`.
- The directory specified by the constant value of variable `P_tmpdir` (usually the root directory of the current drive), which is defined in the operating system header file.
- The current working directory.

Temporary Output

Controls whether temporary files generated during compilation are retained. Permitted settings for this option are:

- **Retain (-keeptempfiles)** — Prevents the deletion of temporary files after they are used. If an assembly language file is created by the compiler, this option will place it in the current directory instead of the temporary directory.
- **Delete (-nokeeptempfiles)** — [default]

Input Languages

Explicitly specifies any languages that the builder might encounter at compile-time. The equivalent driver option is:

- **-language=*language*** — where *language* is one of:
 - **C**
 - **CXX**

When linking or archiving, the driver must know whether or not any files contain other languages in order to select the correct libraries and perform any special processing. If your project includes object files that were compiled from a different language, you must set this option on the object file's parent project or a build item that invokes the linker. This option has no effect when set on an object file.

If your project includes source files that contain other languages and those files appear on the command line, you do not need to set this option.

Non-Standard Output Suffix

Controls the acceptance of non-standard output suffixes. Permitted settings for this option are:

- **Accepted (--any_output_suffix)**
- **Not Accepted (--no_any_output_suffix)** — [default]

Source Directories Relative to Top-Level Project

Specifies a *directory* in which the builder should search for source files. The syntax for this option is:

- **:sourceDirNonRelative**=*directory*

This is a **Builder**-only option. There is no equivalent driver option.

Intermediate Output Directory Relative to This File

Specifies the path to where object files (and any custom file types) are written, relative to the location of the current project **.gpj** file. The syntax for this option is:

- **:outputDirRelative**=*directory*

This option is most useful for custom tools that do not allow you to specify a directory for intermediate output. For projects that use only Green Hills Tools, use **-object_dir** instead. **-object_dir** overrides this option.

This is a Builder-only option. There is no equivalent driver option.

Binary Output Directory Relative to Top-Level Project

Specifies the path to where final executable output files are written, relative to the location of your Top Project (usually **default.gpj**). The syntax for this option is:

- **:binDir**=*directory*

This option controls behavior in the Builder. There is no equivalent driver option.

Binary Output Directory Relative to This File

Specifies the path to where final executable output files are written, relative to the location of the current **.gpj** project file. The syntax for this option is:

- **:binDirRelative**=*directory*

This option controls behavior in the Builder. There is no equivalent driver option.

Dependencies Relative to Source Directory

Specifies that the present file *must* be rebuilt if the specified file, relative to the location of your source directory, changes. The syntax for this option is:

- **:depends=filename**

This option controls behavior in the Builder. There is no equivalent driver option.

Dependencies Relative to This File

Specifies that the present file *must* be rebuilt if the specified file, relative to the location of the present **.gpj** project file, changes. The syntax for this option is:

- **:dependsRelative=filename**

This option controls behavior in the Builder. There is no equivalent driver option.

Dependencies Relative to Top-Level Project

Specifies that the present file *must* be rebuilt if the specified file, relative to the location of your Top Project (usually **default.gpj**), changes. The syntax for this option is:

- **:dependsNonRelative=filename**

This option controls behavior in the Builder. There is no equivalent driver option.

Self-Dependency

Specifies that changes to this Green Hills Project (**.gpj**) file do not cause it or any of its children to be rebuilt. The syntax for this option is:

- **:noSelfDepend**

This option controls behavior in the Builder. There is no equivalent driver option.

Suppress Dependencies

Specifies that the present file *must not* be rebuilt if the specified file changes. The syntax for this option is:

- **:nodepends=filename**

This option controls behavior in the Builder. There is no equivalent driver option.

Output Filename for Generic Types

Specifies a name for the output file of a build item. Use this only for file types that do not accept **-o** or any other option that sets an output name. The syntax for this option is:

- **:outputName=filename**

This option controls behavior in the Builder. There is no equivalent driver option.

Append Default Extension to Output Filename

Appends the default file extension to the output filename if it does not already have an extension, or replaces the existing extension if it does. For example, this option would replace the extension of an object file with **.o** (Linux/Solaris) or **.obj** (Windows). This option is not usually necessary unless you have manually changed the output name of the file. The syntax for this option is:

- **:appendExtension**

This option controls behavior in the Builder. There is no equivalent driver option.

Commands to Execute Before Associated Command

Specifies commands that are executed before the file is processed. The commands are executed immediately before the processing step for the file. If the file has no processing step, the commands are not executed. The following table lists the processing step for common file types:

File Type	Processing Step
C/C++ Source	Compilation
Assembly Source	Assembling
Program	Linking
Library	Archiving
C/C++ Header	None
Project	None

For example:

- If you set **:preExec** on a program, the commands are run after compiling the program, but before linking it.
- If you set **:preExec** on a header file, the commands are not run.

If you are performing a parallel build, use **:preexecSafe** to parallelize the operation.

The syntax for this option is:

- **:preexec=command**

If *command* must include information about the file being processed, use a context sensitive variable (see “Context Sensitive Variables” on page 858).

If you only want to execute the command when building your project in certain contexts, use a conditional control statement (see “Conditional Control Statements” on page 837). For example, to run **preProc.bat** on a file only when building on a Windows host, use the following option:

```
{streq(__MULTI_HOST__, "win32")} :preExec="preProc.bat filename"
```

This option controls behavior in the Builder. There is no equivalent driver option.

Safe Commands to Execute Before Associated Command

Specifies commands that are executed before the file is processed. These commands are guaranteed to be performed before the processing step, but may not be performed immediately before. Commands specified with this option must be *safe*; they must not modify any files related to the build (such as projects, source files, or object files). If you are performing a parallel build, use this option instead of **:preExec** to execute safe commands, because it may speed up the build process. The syntax for this option is:

- **:preexecSafe=command**

For more information about when files are processed, see “**Commands to Execute Before Associated Command**” on page 279.

This option controls behavior in the Builder. There is no equivalent driver option.

Commands to Execute Before Associated Command (via Shell)

Specifies commands to be executed in a shell before the file is processed. For more information about when files are processed, see “**Commands to Execute Before Associated Command**” on page 279. The syntax for this option is:

- **:preexecShell=command**

MULTI prepends the following text to *command* depending on your host’s operating system:

- Windows — *COMSPEC* /c, where *COMSPEC* is the value of the *COMSPEC* environment variable (usually cmd).
- Linux and Solaris — /bin/sh -c

This option controls behavior in the Builder. There is no equivalent driver option.

Safe Commands to Execute Before Associated Command (via Shell)

Specifies commands that are executed in a command prompt or shell before the file is processed. These commands are guaranteed to be performed before the

processing step, but may not be performed immediately before. Commands specified with this option must be *safe*; they must not modify any files related to the build (such as projects, source files, or object files). If you are performing a parallel build, use this option instead of **:preExecShell** to execute safe commands, because it may speed up the build process.

- **:preexecShellSafe=command**

MULTI prepends the following text to *command* depending on your host's operating system:

- Windows — *COMSPEC* /c, where *COMSPEC* is the value of the COMSPEC environment variable (usually cmd).
- Linux and Solaris — /bin/sh -c

This option controls behavior in the Builder. There is no equivalent driver option.

Commands to Execute After Associated Command

Specifies commands that are executed after the file is processed. The commands are executed immediately after the processing step for the file. If the file has no processing step, the commands are not executed. The following table lists the processing step for common file types:

File Type	Processing Step
C/C++ Source	Compilation
Assembly Source	Assembling
Program	Linking
Library	Archiving
C/C++ Header	None
Project	None

For example:

- If you set **:postExec** on an assembly file, the commands are run after assembling the file, but before linking it.
- If you set **:postExec** on a header file, the commands are not run.

If you are performing a parallel build, use **:postexecSafe** to parallelize the operation.

The syntax for this option is:

- **:postexec=command**

If *command* must include information about the file being processed, use a context sensitive variable (see “Context Sensitive Variables” on page 858).

If you only want to execute the command when building your project in certain contexts, use a conditional control statement (see “Conditional Control Statements” on page 837). For example, to run **postProc.bat** on a project’s output file when building on a Windows host, use the following option:

```
{streq(__MULTI_HOST__, "win32")} :postExec="postProc.bat $(OUTPUTFILE) "
```

This option controls behavior in the Builder. There is no equivalent driver option.

Safe Commands to Execute After Associated Command

Specifies commands that are executed after the file is processed. These commands are guaranteed to be performed after the processing step, but may not be performed immediately after. Commands specified with this option must be *safe*; they must not modify any files related to the build (such as projects, source files, or object files). If you are performing a parallel build, use this option instead of **:postExec** to execute safe commands, because it may speed up the build process. The syntax for this option is:

- **:postexecSafe=command**

For more information about when files are processed, see “**Commands to Execute After Associated Command**” on page 281.

This option controls behavior in the Builder. There is no equivalent driver option.

Commands to Execute After Associated Command (via Shell)

Specifies commands to be executed in a command prompt or shell after the file is processed. For more information about when files are processed, see

“**Commands to Execute After Associated Command**” on page 281. The syntax for this option is:

- **:postexecShell**=*command*

MULTI prepends the following text to *command* depending on your host’s operating system:

- Windows — *COMSPEC* /c, where *COMSPEC* is the value of the COMSPEC environment variable (usually cmd).
- Linux and Solaris — /bin/sh -c

This option controls behavior in the Builder. There is no equivalent driver option.

Safe Commands to Execute After Associated Command (via Shell)

Specifies commands that are executed in a command prompt or shell after the file is processed. These commands are guaranteed to be performed after the processing step, but may not be performed immediately after. Commands specified with this option must be *safe*; they must not modify any files related to the build (such as projects, source files, or object files). If you are performing a parallel build, use this option instead of **:postExecShell** to execute safe commands, because it may speed up the build process.

- **:postexecShellSafe**=*command*

MULTI prepends the following text to *command* depending on your host’s operating system:

- Windows — *COMSPEC* /c, where *COMSPEC* is the value of the COMSPEC environment variable (usually cmd).
- Linux and Solaris — /bin/sh -c

This option controls behavior in the Builder. There is no equivalent driver option.

Additional Output Files

Specifies any non-standard output files that are generated during building, so that the **Builder** can include them in file cleanup. The syntax for this option is:

- **:extraOutputFile=filename**

This option controls behavior in the Builder. There is no equivalent driver option.

'Select One' Project Extension List

For use with the Select One project type. Specifies a list of file extensions that determines which file in the project the Builder chooses. The first extension in the list has the highest priority, and the last extension has the lowest. The syntax for this option is:

- **:select=arg1 , arg2 . . .**

For example, if you specify `asm , ppc , c`, the Builder searches your **Select One** project for an `.asm` file first, then a `.ppc` file, and finally, a `.c` file.

If you specify multiple **:select** options for the same project, the builder concatenates them in the order they are specified.

This option controls behavior in the Builder. There is no equivalent driver option.

Pass Through Arguments

Passes unknown options to a command (any tool or utility invoked by the Builder). This is one way of quickly defining custom types. The syntax for this option is:

- **:passThrough=argument**

This option controls behavior in the Builder. There is no equivalent driver option.

Reference

In a **Reference** project file, this option designates the project (**.gpj**) file that is the target of the reference. Create a **Reference** project and use this option when you want to include one project inside of another without carrying over the including project's options. Permitted settings for this option are:

- **:reference=***root_proj*[;*child_proj*]. . . [;*target_proj*]

:reference takes a string argument that is a semicolon-separated list of projects. The Project Manager uses this list as a path to traverse the project build tree and find the project you want to reference (*target_proj*). Each project in the path is the name of a file listed in the build tree, or ". . .", which refers to the parent of the previous project.

If the first project in the list is ". . .", the Project Manager traverses the list beginning at the **Reference** project's parent.

If the first project is the name of a file, the Project Manager traverses the list beginning at the Top Project.

Each project in the list must match the name of a project in the build tree exactly as defined in your project files. If a project's location is defined using a relative or absolute disk path, you must include the disk path in the list. For example, if a project is defined in its parent as **dir/x.gpj**, the **:reference** list item for this project must be **dir/x.gpj**.

This option controls behavior in the Builder. There is no equivalent driver option.

Pattern of Files to Pull Into Project

Specifies the types of files to include in a project. The syntax for this option is:

- **:autoInclude**

For more information about Auto Include project types, see "Automatically Including Files" on page 41.

This option controls behavior in the Builder. There is no equivalent driver option.

Options to Apply to Project

Specifies the name of a file that contains a list of builder or driver options and linker directives (**.ld**) files to use when building a project. This option allows you to share a common set of options and linker directives files among multiple projects. In this file:

- Each line must contain only one option or linker directives file.
- If the line contains an option, it must begin with a whitespace.
- If the line contains a linker directives file, it must begin with the filename.

The lines in this file are processed as if they were present in the **.gpj** file **:optionsFile** is set in. These lines may include macros.

The syntax for this option is:

- **:optionsFile=filename**

This option controls behavior in the Builder. There is no equivalent driver option.

Toolchain Component Locations

This option category is contained within **Advanced→Project Options**.

These options allow you to vary the location of toolchain components.

Alternate Compiler Directory

Specifies an alternate location from which to invoke the compiler. The equivalent driver option is:

- **-Y0,directory**

Alternate Library Directory

Specifies an alternate location to search for the standard system libraries if they are not found in the default location. The equivalent driver option is:

- **-YL,directory**

To search for libraries in another location before searching the default location, use the **-L***directory* option.

If you are not using the default startup code, note that this option changes only the search path for libraries. This means that the linker always takes **crt0.o** from the default location, even if it finds **libstartup.a** and **libsys.a** in another specified location. For more information about how to configure your project to use custom startup code, see “Settings for Stand-Alone Programs” on page 31.

Alternate Assembler Directory

Specifies an alternate location from which to invoke the assembler. The equivalent driver option is:

- **-Y***a,directory*

Alternate Linker Directory

Specifies an alternate location from which to invoke the linker. The equivalent driver option is:

- **-Y***l,directory*

Optimization Options

This option category is contained within **Advanced**.

These options provide very low-level control over various minor optimizations. These optimizations are generally controlled most effectively by the compiler when you specify an **Optimization**→**Optimization Strategy**, and you should not use these options unless you have very particular optimization needs. For more common options, see “Optimization Options” on page 176.

Inline Tiny Functions

Controls the inlining of very small functions. Permitted settings for this option are:

- **On** (**--inline_tiny_functions**) — This setting is implied by an **Optimization**→**Optimization Strategy** setting of **Optimize for Speed**

(**-Ospeed**), **-Osize**, or **-Ogeneral** unless **Debugging**→**Debugging Level** is set to **-g** or **-G**.

- **Off** (**--no_inline_tiny_functions**) — [default]

For more information, see “Automatic Inlining” on page 317.

Inline C Memory Functions

Controls the inlining of C Memory Functions. Permitted settings for this option are:

- **On** (**-Omemfuncs**)
- **Off** (**-Onomemfuncs**) — [default]

For more information, see “Inlining of C Memory Functions” on page 325.

Inline C String Functions

Controls the inlining of C String Functions. Permitted settings for this option are:

- **On** (**-Ostrfuncs**)
- **Off** (**-Onostrfuncs**) — [default]

For more information, see “Inlining of C String Functions” on page 326.

Simplify C Print Functions

Controls print function optimization. You may need to set this option to **off** if you define your own print functions (such as `fprintf()`, `fputs()`, `printf()`, `puts()`, etc.) Permitted settings for this option are:

- **On** (**-Oprintfuncs**) — [default] Enable print function optimizations.
- **Off** (**-Onoprintfuncs**) — Disable print function optimizations.

Loop Unrolling

Controls the *Loop Unrolling* optimization. Permitted settings for this option are:

- **On** (-Ounroll)
- **Off** (-Onounroll)

For more information, see “Loop Unrolling” on page 342.

Register Allocation by Coloring

Controls the dynamic storage of variables in registers. Permitted settings for this option are:

- **On** (-overload) — [default]
- **Off** (-nooverload)

For more information, see “Register Allocation by Coloring” on page 362.

Automatic Register Allocation

Controls the automatic allocation of local variables to registers. Permitted settings for this option are:

- **On** (-autoregister) — [default]
- **Off** (-noautoregister)

For more information, see “Automatic Register Allocation” on page 363.

Common Subexpression Elimination

Controls the *Common Subexpression Elimination* optimization. Permitted settings for this option are:

- **On** (-Ocse)
- **Off** (-Onocse)

For more information, see “Common Subexpression Elimination/Partial Redundancy Elimination” on page 355.

Tail Calls

Controls the *Tail Recursion* optimization. Permitted settings for this option are:

- **On** (-Otailrecursion)
- **Off** (-Onotailrecursion)

For more information, see “Tail Calls” on page 356.

Constant Propagation

Controls the *Constant Propagation* optimization. Permitted settings for this option are:

- **On** (-Oconstprop)
- **Off** (-Onoconstprop)

For more information, see “Constant Propagation” on page 358.

C/C++ Minimum/Maximum Optimization

Controls the *C/C++ Minimum/Maximum* optimization. Permitted settings for this option are:

- **On** (-Ominmax)
- **Off** (-Onominmax)

For more information, see “C/C++ Minimum/Maximum Optimization” on page 359.

Memory Optimization

Controls the K+R C *Memory* optimization. If no optimization strategy is selected, this option enables **-Ogeneral**. Permitted settings for this option are:

- **On** (-OM)
- **Off** (-Onomemory) — [default]

For more information, see “Memory Optimization” on page 359.

Optimize All Appropriate Loops

Turns on all the **General Use** optimizations, together with loop strength reduction, loop invariant analysis, and loop unrolling. Permitted settings for this option are:

- **On (-OL)**
- **Off (-Onoloop)** — [default]

For more information, see “Loop Optimizations (including SIMD Vectorization)” on page 339.

Peephole Optimization

Controls peephole optimizations. Permitted settings for this option are:

- **On (-Opeep)**
- **Off (-Onopeephole)**

For more information, see “Peephole Optimization” on page 351.

Pipeline Instruction Scheduling

Controls pipeline optimizations. Permitted settings for this option are:

- **On (-Opipeline)**
- **Off (-Onopipeline)**

For more information, see “Pipeline Instruction Scheduling” on page 352.

GNU Compatibility Optimizations

Provides compatibility with the general GNU optimization options. Permitted settings for this option are:

- **O1 (-O1)** — Turns on all the **-Ogeneral** optimizations.
- **O2 (-O2)** — Turns on all the **-O1** optimizations, loop strength reduction, and loop invariant analysis. For further details, see “Loop Optimizations (including SIMD Vectorization)” on page 339.

- **O3 (-O3)** — Turns on all the **-O2** optimizations and two-pass inlining. For further details, see “Inlining Optimizations” on page 317.
- **O0 (-O0)** — [default] No optimizations.

Vector Peeling

Controls whether the compiler can perform loop peeling while vectorizing loops. Permitted settings for this option are:

- **On (-Ovectorpeel)**
- **Off (-Onovectorpeel)**

Delay Second Pass

Permitted settings for this option are:

- **On (-delay_second_pass)** —
- **Off (-nodelay_second_pass)** —

Delay Archives

Permitted settings for this option are:

- **On (-delay_archives)** —
- **Off (-nodelay_archives)** —

Package Analysis Files

Permitted settings for this option are:

- **On (-package_infs)** —
- **Off (-nopackage_infs)** —

Debugging Options

This option category is contained within **Advanced**.

These options provide control over advanced debugging features. For more common options, see “Debugging Options” on page 187.

Generate Target-Walkable Stack

Forces the compiler to use a stack convention that allows the program to walk its own stack without using debugging information. However, optimizations might cause gaps in the stack trace.

Permitted settings for this option are:

- **On** (**-gtws**)
- **Off** (**-nogtws**) — [default]

-Olimit= Without Debug Information

Instructs the compiler to limit optimizations for debugging, even when debug information is not being generated. By default, **-Olimit=peephole** and **-Olimit=pipeline** are only meaningful when debug information is enabled. However, the compiler can still make some attempt to restrict optimizations to assist debugging, even when debug information is never generated. Note that the results will not be as good as they would be if debug information were enabled. Permitted settings for this option are:

- **On** (**-glimits**)
- **Off** (**-noglimits**) — [default]

Consistent code without debug information

Instructs the compiler to produce roughly the same code with the **--no_debug** option as it does with either **-g** or **-G**, although the code is not guaranteed to be identical. Do not use this option with **-g**, **-gs**, or **-G**. Permitted settings for this option are:

- **On** (**-consistentcode**)
- **Off** (**-noconsistentcode**) — [default]

MULTI Version

Specifies the version of MULTI that will be used to debug code generated by the compiler. The equivalent driver option is:

- **--multi_version=version**

Where *version* is one of the following values:

- **v40** — MULTI 4.x.
- **v50** — MULTI 5.x.
- **v60** — MULTI 6.x.

Full Breakdots

Controls the number of breakdots available in the MULTI Debugger. This option has an effect only when **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug** is enabled. Permitted settings for this option are:

- **On (-full_breakdots)** — Increases the number of breakdots available in the MULTI Debugger, but may add additional instructions to your code. This is the default setting if **-Omaxdebug** or **-Omoredebug** is enabled.
- **Off (-no_full_breakdots)** — This is the default setting if **-Onone** or **-Odebug** are enabled.

Extend Variable Liveness

Controls access to local automatic variables in the MULTI Debugger. This option has an effect only when **-Onone**, **-Odebug**, **-Omoredebug**, or **-Omaxdebug** is enabled. Permitted settings for this option are:

- **On (-extend_liveness)** — Local automatic variables are accessible from the MULTI Debugger after they are written. This is the default setting if **-Omaxdebug** or **-Omoredebug** is enabled.
- **Off (-no_extend_liveness)** — Local automatic variables are accessible from the MULTI Debugger as long as they might be read before being written to. This is the default setting if **-Onone** or **-Odebug** are enabled.

Search Path for .dbo Files

Specifies a *directory* to search for **.dbo** debugging information files. The equivalent driver option is:

- **-dbopath** *directory*

This option is only needed if object files are moved between being compiled and linked.

Search for DBA

Controls searching for **.dba** files in incremental link situations and enables creation of a **.dba** file corresponding to the **.o** file created by an incremental link.

If you want to generate debug information for a relocatable object file (using **-G** and **-relobj**) and the machine that will be performing the subsequent link will not have access to the original **.o** or **.dbo** files, pass the **-search_for_dba** option.

For example, the following set of options creates a **myobj.dba** file that you can distribute with **myobj.o** to provide debug information to MULTI:

```
ccppc file1.c file2.c -G -relobj -o myobj.o
```

When performing subsequent links with **myobj.o**, continue to pass the **-search_for_dba** option to force **dblink** to search for **myobj.dba** instead of only looking for **myobj.dbo**.

-search_for_dba is not necessary if the original **.dbo** files that comprise **myobj.dba** (**file1.dbo** and **file2.dbo** in the above example) are still available.

Permitted settings for this option are:

- **On** (**-search_for_dba**)
- **Off** (**-no_search_for_dba**) — [default]

Force Frame Pointer

Controls the use of a frame pointer during code generation. Permitted settings for this option are:

- **On** (**-ga**) — Always use a frame pointer.

- **Off (-noga)** — [default] Use of a frame pointer is not guaranteed.

Scan source files to augment native debug info

Improves the code browsing features of MULTI when using code compiled by third-party compilers. This feature causes the **dwarf2dbo** or **stabs2dbo** Debug Translator to gather information directly from source files. **--scan_source** causes source scanning to occur whenever debugging information is translated as part of the project build. Permitted settings for this option are:

- **On (--scan_source)**
- **Off (--no_scan_source)**

You can use **--scan_source** if the root of your sources has changed. For more information about **--scan_source**, see “Building and Debugging on Different Hosts” in Appendix D, “Using Third-Party Tools with the MULTI Debugger” in the *MULTI: Debugging* book, and “Source Scanning” in Appendix D, “Using Third-Party Tools with the MULTI Debugger” in the *MULTI: Debugging* book.

Add Extra File to Debug Info

When this option is used, an entry for the specified file is added to the debug information generated by the compiler. This allows the debugger to find that file when you click its name or use the **e** command. This option is useful for including notes or text-based custom files in to the set of files the debugger can display. By default, the builder (command line or graphical) includes all source code and "Text" files in the debug information. The equivalent driver option is:

- **-extra_file=filename**

Set Maximum DBO Not Found Warnings

Limits the number of "dbo not found" warnings to the given number. The default limit is 9. Setting the limit to 0 hides all such warnings. The equivalent driver option is:

- **-warn_dbo_not_found_max=n**

Wait for Debug Translation Before Exiting

Controls whether the compiler driver waits for debug information to be generated before exiting. The program can be executed immediately after the driver exits, but debugging is not possible until **dblink** has finished generating debug information. Permitted settings for this option are:

- **On (-wait_for_dblink)** — [default] The compiler driver waits for **dblink** to finish generating debug information before exiting.
- **Off (-do_not_wait_for_dblink)** — The compiler driver does not wait for **dblink**. This option is not supported with **-strip**. When using this option, do not open MULTI or other executables that require debug information until debug translation has completed.

Reducing Debug File Size

This option category is contained within **Advanced→Debugging Options**.

These options allow you to vary the amount of debugging information that will be generated for use with the MULTI Debugger. Note that reducing the amount of information collected may speed up performance, but it may also reduce the depth of analysis available through the Debugger.

Cross-Reference Information

Controls the amount of debugging cross-referencing information that is generated. By default, the compiler generates cross-reference data for each user-defined construct (describing where the construct is declared, defined, read from, written to, and so on), to allow the Debugger to display cross references. Permitted settings for this option are:

- **Generate for All Objects (--xref=full)** — [default] Generates cross-referencing data for all objects.
- **Generate for Object Declarations and Definitions (--xref=declare)** — Generates cross-referencing data only for the declarations and definitions of the objects.
- **Generate for Objects in the Global Scope (--xref=global)** — Generates cross-referencing data for the objects that are declared in the global scope.
- **Do Not Generate (--xref=none)** — Disables all cross-referencing data.

Debugging Information

Controls the generation of debugging information for types and defines that are never used in a file. Permitted settings for this option are:

- **All (-full_debug_info)** — Enables the generation of debugging information for types and defines that are never used in a file.
- **Reduced (-no_full_debug_info)** — [default] Disables the generation of debugging information for types and defines that are never used in a file.

Native Debugging Information

This option category is contained within **Advanced→Debugging Options**.

These options control the generation of alternate native debugging formats. Use them only if you use third-party tools that require native debugging information. The MULTI Debugger only uses the **.dbo** format.

Generate MULTI and Native Information

Enables the generation of DWARF, COFF, or BSD debugging information in the object file (in addition to the Green Hills **.dbo** format), according to the convention for your target. Permitted settings for this option are:

- **On (-dual_debug)**
- **Off (-no_dual_debug)** — [default]

Requires that the **Debugging→Debugging Level** be set to Plain or MULTI for the option to have any effect (see “Debugging Options” on page 187).

Convert DWARF Files to MULTI (.dbo) Format

Controls the conversion of DWARF debugging information to the MULTI **.dbo** format. Permitted settings for this option are:

- **On (-dwarf2dbo)**
- **Off (-no_dwarf2dbo)** — [default]

This option will normally be invoked automatically when necessary by the builder.

Convert Stabs Files to MULTI (.dbo) Format

Controls the conversion of Stabs debugging information to the MULTI **.dbo** format. Permitted settings for this option are:

- **On** (**-stabs2dbo**)
- **Off** (**-no_stabs2dbo**) — [default]

This option will normally be invoked automatically when necessary by the builder.

Preprocessor Options

This option category is contained within **Advanced**.

These options control advanced preprocessor settings. For more common options, see “Preprocessor Options” on page 192.

Process #include Directives

Controls the treatment of duplicate `#include` statements. These options are deprecated and may be removed in a future MULTI release. Permitted settings for this option are:

- **Ignore #include directives** (**--include_never**) — Ignore all `#include` directives.
- **Process each file only once** (**-include_once**) — Includes a file only once, even if it is referenced in multiple `#include` statements.
- **Process #includes normally** (**-no_include_once**) — [default] Includes a file every time it is referenced.

Some third party header files expect a file to be included multiple times (for instance Linux C header files). In these environments, you should not set this option to **Prevent** (**-include_once**). The compiler will still automatically skip those files which have include macro guards, even without this option.

(Windows) This option does not prevent a file from being included twice if the file is specified using a different pathname. For example, the compiler will treat **test.h** as two different files if specified like this on Windows:

```
#include "test.h"
#include "sub/../test.h"
```

(Linux/Solaris) The compiler detects that both `#include` directives refer to the same file.

Files to Pre-Include

Includes the source code of the specified *filename* at the beginning of the compilation. The equivalent driver option is:

- **-include** *filename*

This can be used to establish standard macro definitions, and so on. The *filename* is searched for in the directories on the include search list.

Definition of Standard Symbols

Controls the definition of the set of default symbols. Permitted settings for this option are:

- **Define (-stddef)** — [default] You can use this option in a MULTI Project (.gpj) file, but there is no equivalent driver option.
- **Do Not Define (-nostddef)**

Definition of Unsafe Symbols

Controls the predefinition of preprocessor macros without leading underscores. Permitted settings for this option are:

- **Define (--unsafe_predefines)** — Predefine macros both with and without leading underscores.
- **Do Not Define (--no_unsafe_predefines)** — [default] Predefine macros only with leading underscores.

When compiling C++ code in Strict Standard Mode, this option has no effect.

Retain Comments During Preprocessing

Controls the retention of comments by the preprocessor. Permitted settings for this option are:

- **Retain (-C)**
- **Strip (--no_comments)** — [default]

Message Pragma

Controls the use of #pragma message in C and C++ source files. Permitted settings for this option are:

- **Accept pragma message and emit message (--pragma_message)** — [default] Accept #pragma message directives and print the string to standard error output while compiling. The -w option does not suppress messages produced by the directive. When --stop-on-warnings is used, compilation does not stop if a message is printed.
- **Accept pragma message and suppress message (--pragma_message_silent)** — Accept #pragma message directives with any format, including those that would be rejected by --pragma_message, and ignore the pragma.
- **Do not accept pragma message (--pragma_message_unknown)** — Treat #pragma message directives as an unknown pragma directive. See --unknown_pragma_warnings.

Advanced Include Directories

This option category is contained within **Advanced→Preprocessor Options**.

By default, the Builder and driver search for header files in various standard directories provided with your distribution. These options allow you to vary the location of these standard directories.

Standard Include Directories

Controls whether the standard include directories are searched. Permitted settings for this option are:

- **Search (-stdinc)** — [default]
- **Do Not Search (-nostdinc)**

C Include Directories

Changes the default location in which the compiler searches for C header files. The equivalent driver option is:

- **-c_include_directory** *directory*

C++ Include Directories

Specifies a directory in which the builder should search for C++ header files. The equivalent driver option is:

- **--cxx_include_directory** *directory*

C++ Standard Include Directories

Changes the default location in which the compiler searches for C++ header files. The equivalent driver option is:

- **-std_cxx_include_directory** *directory*

System Include Directories

Changes the default location in which the compiler searches for system header files. The equivalent driver option is:

- **-sys_include_directory** *directory*

Source Directory Is Include Directory

Controls whether the directory of the source file is searched for include files when using **-I-**. Permitted settings for this option are:

- **On (--search_source_directory)** — When used with **-I-**, search the directory of the source file for include files.

- **Off (--no_search_source_directory)** — [default] When used with **-I-**, do not search the directory of the source file for include files.

C/C++ Compiler Options

This option category is contained within **Advanced**.

These options control advanced C and C++ compilation settings. For more common options, see “C/C++ Compiler Options” on page 194.

C++ Inlining Level

Controls the inlining of C++ functions. Permitted settings for this option are:

- **Maximum (--max_inlining)** — Considers all appropriate functions for inlining.
- **Maximum Unless Debugging (--max_inlining_unless_debug)** — Performs maximum inlining unless debugging information is being collected.
- **Standard (--inlining)** — [default] Considers only relatively small functions without control flow statements for inlining.
- **Standard Unless Debugging (--inlining_unless_debug)** — Performs standard inlining unless debugging information is being collected.
- **None (--no_inlining)** — Disables inlining of C++ functions. This option also disables **--inline_tiny_functions**, **-OI**, **-Oipsmallinlining**, and **-Oiponesiteinlining**.

For more information, see “Additional C++ Inlining Information” on page 328.

Emulated GNU Version

Controls the version of the GNU compiler that the Green Hills compiler provides compatibility with. Permitted settings for this option are:

- **GNU 3.3 (--gnu_version=30300)** — When using a GNU dialect, provide close compatibility with the GNU 3.3 compiler.
- **GNU 4.3 (--gnu_version=40300)** — [default] When using a GNU dialect, provide close compatibility with the GNU 4.3 compiler. However, this

option does not enable **C/C++ Compiler→C++→Templates→Dependent Name Processing (--dep_name)** by default.

- **GNU 4.3 (strict) (--gnu_version=40300_strict)** — When using a GNU dialect, provide close compatibility with the GNU 4.3 compiler. When using a GNU dialect, this option enables **C/C++ Compiler→C++→Templates→Dependent Name Processing (--dep_name)** and **C/C++ Compiler→C++→Templates→Deferral of Function Template Parsing (--defer_parse_function_templates)** by default, which is more similar to the GNU 4.3 compiler but enabling dependent name processing is also more strict, as non-dependent names are looked up when a template is parsed.

Prepend String to Every Section Name

Specifies a *string* to prepend to every section name. The equivalent driver option is:

- **--section_prefix** *string*

Append String to Every Section Name

Specifies a *string* to append to every section name. The equivalent driver option is:

- **--section_suffix** *string*

longjmp() Does Not Restore Local Vars

Controls whether the values of local non-volatile variables are modified by `longjmp` calls. ANSI C standards require only that the values of volatile variables are not modified. Protecting non-volatile values requires additional overhead. Permitted settings for this option are:

- **On (-locals_unchanged_by_longjmp)** — Guarantees that the values of all local variables are not modified by `longjmp` calls.
- **Off (-no_locals_unchanged_by_longjmp)** — [default] Guarantees only that the values of volatile local variables are not modified by `longjmp` calls.

Accept GNU `__asm__` statements

Provides a way to enable the GNU extended `asm` syntax support. The `-gcc` and `--g++` options enable this support, but also enable other extensions. Permitted settings for this option are:

- **On** (`--gnu_asm`) — Enables GNU extended `asm` syntax support.
- **Off** (`--no_gnu_asm`) — [default] The `-gcc` or `--g++` option can be used to enable this support even when `--no_gnu_asm` is passed.

For a full description of the syntax, see the GNU manuals. The following modifiers are accepted:

=	Operand is write only for this statement.
+	Operand is read and written by this statement.
&	Operand is modified before all input operands are read by this statement.

The target independent constraints accepted are:

x	Any operand is allowed.
g	Any general register, memory, or immediate operand is allowed.
r	A general register operand is allowed.
f	Any operand is allowed.
m	A memory operand is allowed.
p	A memory address operand is allowed.
o	A memory operand with an offset is allowed.

V	A memory operand without an offset is allowed.
<	A decrementing memory operand is allowed.
>	An incrementing memory operand is allowed.
i	An immediate integer operand is allowed.
n	An immediate integer operand with a known numeric value is allowed.
s	An immediate integer operand without a known numeric value is allowed.
E	An immediate floating point operand is allowed.
F	An immediate floating point operand is allowed.

The following Power Architecture specific constraints are accepted:

b	An address base register is allowed, excluding <code>r0</code> , <code>r1</code> , <code>r2</code> , and <code>r13</code> .
h	The <code>MQ</code> , <code>CTR</code> , or <code>LINK</code> registers are allowed.
q	The <code>MQ</code> register is allowed.
c	The <code>CTR</code> register is allowed.
l	The <code>LINK</code> register is allowed.
x	The <code>CR</code> register number zero is allowed.
Y	The <code>CR</code> register is allowed.

Z	The <code>XER[CA]</code> carry bit is allowed.
I	A signed 16 bit constant operand is allowed.
J	An unsigned 16 bit constant operand shifted left 16 bits is allowed.
K	An unsigned 16 bit constant operand is allowed.
L	A signed 16 bit constant operand shifted left 16 bits is allowed.
M	A constant larger than 31 is allowed.
N	An exact power of 2 is allowed.
O	The constant zero is allowed.
P	A constant whose negation is a signed 16 bit constant is allowed.
G	A floating point constant that can be loaded with one instruction is allowed.
Q	A memory operand that includes a register and offset is allowed.
T	A constant suitable as a 32 bit mask is allowed.
U	A small data area reference operand is allowed.

Advanced C++ Options

This option category is contained within **Advanced→C/C++ Compiler Options**.

These options control advanced C++ compilation settings.

Demangling of Names in Linker Messages

Controls the demangling of names that appear in linker messages. These are typically symbol names that are either undefined or multiply defined. Permitted settings for this option are:

- **On** (`--link_filter`) — [default]
- **Off** (`--no_link_filter`)

C++ Linking Method

Controls the method for generating constructors/destructors at link-time. Permitted settings for this option are:

- **Use Green Hills Linker** (`--linker_link`) — [default]
- **Create Array of Static Constructors/Destructors for Post-Link Phase** (`--munch`)
- **Do not Generate Constructors/Destructors** (`--nocpp`)

Distinct C and C++ Functions

Controls whether identical C and C++ functions are treated as distinct. Permitted settings for this option are:

- **On** (`--c_and_cpp_functions_are_distinct`) — Treat function types as distinct if their only difference is that one has `extern "C"` linkage and the other has `extern "C++"` routine linkage.
- **Off** (`--no_c_and_cpp_functions_are_distinct`) — [default] Do not treat such functions as distinct.

Support for Implicit Extern C Type Conversion

Controls an extension to permit implicit type conversion in C++ between a pointer to an `extern C` function and a pointer to an `extern C++` function. Also, see “**Distinct C and C++ Functions**” on page 308. Permitted settings for this option are:

- **On** (`--implicit_extern_c_type_conversion`)
- **Off** (`--no_implicit_extern_c_type_conversion`) — [default]

Assembler Options

This option category is contained within **Advanced**.

These options control advanced assembler settings. For more common options, see “C/C++ Compiler Options” on page 194.

Assembler Warnings

Controls the display of assembler warning messages. Permitted settings for this option are:

- **Display** (`--assembler_warnings`) — [default]
- **Suppress** (`--no_assembler_warnings`)

Identifier Definition

Passes the arbitrary string *string* to the output file. The equivalent driver option is:

- **-ident**=*string*

This is the same as using the directive `#pragma ident "string"` in C (see “General Pragma Directives” on page 680). This option can be used to place the date of the source file in the object file.

Identifier Support

Controls support for identifier definition and insertion (via the Builder and driver options above, and through `#pragma ident "string"`). Permitted settings for this option are:

- **On** (`-identoutput`) — [default]
- **Off** (`-noidentoutput`)

Linker Options

This option category is contained within **Advanced**.

These options control advanced linker settings. For more common options, see “Linker Options” on page 244.

Support floating point types in scanf

Controls the inclusion of floating-point `scanf()` code. Permitted settings for this option are:

- **Off (-no_float_scanf)** — Inform the compiler and linker that no calls to `scanf()` or its related functions, such as `fscanf()` or `sscanf()`, use floating point, therefore it is not necessary to bring in the support code for floating-point `scanf()`, saving space in the target program. This option is implied by **-fnone**.
- **On (-float_scanf)** — [default] Bring in the floating-point support code for `scanf()` if there is any call to `scanf()` or its related functions.

Linker Directives Directory

Specifies the directory containing the default linker directives file or files. This option has no effect if you explicitly specify a linker directives file in your project or on the command line. The equivalent driver option is:

- **-directive_dir=directory**

Link in Standard Libraries

Controls the linking-in of any Green Hills standard startup files or libraries. User-specified libraries are not affected. Permitted settings for this option are:

- **Link (-stdlib)** — [default] Use Green Hills standard libraries.
- **Do Not Link (-nostdlib)** — Do not use Green Hills standard libraries, or link against the Green Hills or any user-defined start or end files (see “Start and End Files” on page 252).

When you use this option, the linker may return an unresolved symbol error due to references to architecture support routines, even if your code has no apparent external library dependencies.

Force Deletion of Unused C++ Virtual Function Symbol

Specifies C++ virtual function symbols that should be deleted even if they appear to be used. The equivalent driver option is:

- **-force_uvfd=***symbol*

To specify more than one symbol, use this option multiple times.

Link Mode

Controls overwriting of an existing executable. By default, the linker overwrites any existing file with the same name as the specified output file. However, the driver can also save a temporary backup copy of the existing file in order to restore it if the link fails. If the link succeeds, the backup copy is silently deleted. Permitted settings for this option are:

- **Overwrite the existing executable (--link_output_mode_reuse)** — [default]
- **Save a temporary copy unless the existing file is a symlink (--link_output_mode_safe)**
- **Save a temporary copy, even if the existing file is a symlink (--link_output_mode_linksafe)**
- **Remove the existing executable before writing a new version (--link_output_mode_unlink)**

Linking Sections with Different Types

Allows the linker to combine and link sections from different object files that have different section types (for example, SHT_PROGBITS and SHT_NOBITS). Without this option, the linker will provide an error message and not complete the link. The equivalent driver option is:

- **--allow_different_section_types**

Physical Address Offset From Virtual Address

Instructs the linker to make `p_paddr` (the physical address) in the ELF program headers equal to `p_vaddr` (the virtual address) plus this offset, *n*.

Specifying 0 for the offset makes `p_paddr` equal to `p_vaddr`. The equivalent driver option is:

- **-paddr_offset=*n***

Follow Section

The equivalent driver option is:

- **-follow_section *x=y***
- **-follow_section "*x=y* [*start_expression*] [*attributes*] : [{ *contents* }] [>*memname* | >.]"**

Inserts a definition for section *y* into the section map immediately following the definition of section *x*. The *start_expression*, *attributes*, and *contents* of section *y* may be specified explicitly, using standard linker directives file syntax. If section *x* has the `ABS`, `CLEAR`, or `NOCLEAR` attributes, section *y* inherits them. If the section map contains a section *z* with the attribute `ROM(x)`, `CROM(x)`, or `ROM_NOCOPY(x)`, then a new definition for the section `.ROMy`, `.CROMy`, or `.ROM_NOCOPYy` is inserted following *z* as well.

For more information about these attributes and linker directives file syntax, see Chapter 11, “The elxr Linker”.

Compiler Diagnostics Options

This option category is contained within **Advanced**.

Redirect Error Output to Overwriting File

Redirects error output to a file, overwriting any existing file, unlike **-stderr=** which just appends output to a file. The equivalent driver option is:

- **-stderr_overwrite=*file***

Because the overwriting takes effect when compiling each individual source file, **-stderr_overwrite** is probably only desired in a makefile situation where the `stderr` output filename is named after the input filename. For example:

```
ccppc file1.c -stderr_overwrite=file1.err -c  
ccppc file2.c -stderr_overwrite=file2.err -c
```

as opposed to how you might normally use **-stderr**:

```
ccppc file1.c -stderr=err.out -c  
ccppc file2.c -stderr=err.out -c
```

Support Diagnostics Options

This option category is contained within **Advanced**.

You should use these options only if instructed to do so by Green Hills support staff.

X-Switches

Specifies one or more internal compiler switches. The equivalent driver option is:

- **-Xswitch**

EDG Front End Options

Specifies one or more internal EDG front end options. The equivalent driver option is:

- **--option=option**

Debug Information (.dbo) Tracing Diagnostics

Controls the collection of diagnostic information pertaining to a search for **.dbo** files. Permitted settings for this option are:

- **On (-dbo_trace)**

This option is useful if you are seeking support for an incident involving missing **.dbo** files.

Dbo File Version

Specifies an older version of **.dbo** debugging information files, for use with an old MULTI Debugger. The equivalent driver option is:

- **--dbo_version=*n***

HTML Compiler Options

This option category is contained within **Advanced**.

This options in this section are deprecated and provided for backwards compatibility only.

Alternate Header Name

This option is deprecated and provided for backwards compatibility only.

Renames the generated include file. The equivalent HTML compiler option is:

- **-h**

Optimizing Your Programs

Chapter 7

This Chapter Contains:

- Optimization Descriptions

Green Hills provides many different ways of optimizing code, both to reduce the size and to increase the speed of your program. The simplest way to enable optimizations is to set an optimization strategy, which groups several optimizations into one setting. To enable general optimizations:



Set the **Optimization→Optimization Strategy** option to **Optimize for General Use (-Ogeneral)**.

To enable optimizations that emphasize the reduction of program size over performance speed:



Set the **Optimization→Optimization Strategy** option to **Optimize for Size (-Osize)**.

To enable optimizations that emphasize the increase of performance speed over program size:



Set the **Optimization→Optimization Strategy** option to **Optimize for Speed (-Ospeed)**.

To attempt to further increase your program speed (at the cost of a substantial increase in compilation time), set the **Optimization→Intermodule Inlining** option to **On (-OI)**



Note Enabling debugging information with the **-g** or **-G** options disables some optimizations in order to make code easier to debug.

For information about these and other optimization options, see “Optimization Options” on page 176. For detailed descriptions of various optimizations, see the following section.

Optimization Descriptions

This chapter provides detailed descriptions of many of the optimizations available with the Green Hills PowerPC compiler. It is provided for users who are interested in detailed information about the code transformations that these

optimizations perform, and for advanced users who may want to control them individually.

Inlining Optimizations

The term *inlining* refers to the process of substituting a call to a function or subroutine with the actual content of that function or subroutine. This eliminates the overhead of a subroutine call and may result in faster code.

Typically, the best candidate for inlining is a small, frequently executed function or subroutine that the program calls from only a few locations. Inlining can then increase efficiency in high usage areas without significantly increasing program size.

The Green Hills implementation of inlining is language independent within the Green Hills family of compilers. Routines of one language may be freely inlined into programs of another language. Additionally, the Green Hills compilers perform inlining across modules. Therefore, if a program defines a function within one module, and several modules use that same function, the compiler can inline the function within all the appropriate modules.

What Can Be Inlined

Although you can request that any number of functions be considered for inlining using the methods described below, the compiler will not necessarily inline all such functions. The overall size of the function to be inlined, the calling function, and other factors are always taken into consideration.

Automatic Inlining



This option is enabled by default with all optimizations (**-Osize**, **-Ospeed**, and **-Ogeneral**), unless debug information is set. Additionally, the optimization can be enabled explicitly with **--inline_tiny_functions**.

To disable this optimization, set the **Advanced→Optimization Options→Inline Tiny Functions** option to **Off** (**--no_inline_tiny_functions**).

The compiler may automatically inline the following functions:

- Static functions that a file references only one time.
- Very small functions (for example, single simple assignments).

Manual Inlining

This section explains how to manually suggest functions for inlining. You cannot force the compiler to inline any function.

Manual Inlining In C

To manually indicate that the compiler should consider a function for inlining, place an inline keyword immediately before the declaration of the function. A list of inline keywords is provided in the table at the end of this section.

If the compiler decides that a function declared with an inline keyword cannot be inlined, it must either create an out-of-line copy in the module or externally link to another module's out-of-line copy. There are four different methods the toolchain uses to link these functions, depending on how the function was declared. The table at the end of this section describes the declarations, and the following list describes the methods:

- *Exported Inline* — The compiler always generates an out-of-line copy of the function, even if it is not required by the module. If you have two exported inline functions with the same name, the linker will report that you have multiply defined symbols.
- *Imported Inline* — The compiler never generates an out-of-line copy of the function. It always assumes that a copy exists in another module. If you declare an imported inline function and there is no exported inline definition of the function in any translation unit, the linker may report that you have an undefined symbol.
- *Static Inline* — The compiler generates a normal static function only when it is required by the module (for example, if you take the address of the function or if the function is recursive). While you will not get linker errors with static inline functions, there are some important consequences:
 - If you compare two static inline functions in different modules, they will not equate because they have different addresses.
 - Your code size might increase, because there might be multiple out-of-line copies of the function.

If you want to manually inline functions, we recommend that you use static inlining whenever possible. Static inlined functions are portable, do not cause linker errors, and behave exactly the same as static functions when they cannot be inlined. If you want to explicitly control the linkage of one of these functions to decrease code size:

- Make one exported inline definition of the function.
- For all other inline declarations, make imported inline declarations.
- If you do not want to inline the function in a certain translation unit, provide a non-inline external declaration in the translation unit. For example:

```
/* do not inline the function in this translation unit */
extern int inline_max(int, int);
```

The following table lists which linking behavior the compiler uses for inline functions based on your C language dialect and the keyword you use when declaring the function.

Keyword	Dialect	
	GNU	C89/C99
<code>inline</code>	exported	imported
<code>__inline</code>	exported	static
<code>__inline__</code>	exported	static
<code>extern inline</code>	imported	exported
<code>extern __inline</code>	imported	static
<code>extern __inline__</code>	imported	static
<code>static inline</code>	static	static

Manual Inlining in C++

To manually indicate that the compiler should consider a C++ function for inlining, place the `inline` keyword immediately before the declaration of the function. Member functions that are defined inside class or `struct` type definitions are also considered inline, even if you do not declare the function with an inline keyword. Such functions are known as *implicitly inline member functions*.

In C++, the compiler generates an out-of-line copy of the function only when it is required by the module (for example, if you take the address of the function

or if the function is recursive). Static data within the function is shared among all copies. While you will not get linker errors if you internally inline functions, there are some important consequences:

- If you compare two functions in different modules, they will not equate because they have different addresses.
- Your code size might increase, because there might be multiple out-of-line copies of the function.

If you want force the compiler to use just one out-of-line copy of an inline function for the whole program, use the `--instantiate_extern_inline` option and declare that function with `extern inline` (see “**Instantiate Extern Inline**” on page 231).

Single-pass Inlining

This optimization is enabled by default and cannot be disabled.

Only those functions indicated by the user in the source code will be considered for inlining.

Example 1. Inlining Principles

The following program illustrates the basic principles of inlining. The main program in this case contains a simple loop that calls the function `sub()`. The call itself occurs only once in the program code, but the function is executed for each iteration of the loop. The call is easily replaced by the code for `sub()` itself, eliminating both the need for parameter passing and the overhead of a jump-to-subroutine. The reduced overhead per execution results in an increase in program speed.

Initial C source code	Optimized C source code
<pre> __inline void sub(int x) { printf("x=%d\n",x); return; } int main() { int i; for (i=1;i<10;i++) sub(i); return 0; } </pre>	<pre> int main() { int i; for (i=1;i<10;i++) printf("x=%d\n",i); return 0; } </pre>

Two-Pass Inlining (Intermodule Inlining)

Two-pass inlining involves a special precompilation pass over all of the source files to generate intermediate code for all of the functions to be inlined. This code is stored in files with the **.inf** extension. On the second pass, each source file is compiled normally, but the compiler is given all of the **.inf** files as additional input. In this way, the compiler is able to inline functions across source files and across programming languages.

In two-pass inlining, compiler warnings are suppressed on the first pass so that duplicate warnings will not be issued.

Selecting All Small Functions for Two-Pass Inlining



To control this optimization, use the **Optimization→Intermodule Inlining** option (**-OI/-Onoinline**).

This optimization instructs the compiler to consider most small functions as candidates for inlining. You do not need determine which functions to inline, or modify the source code.

Any functions that have been manually specified will also be considered for inlining.

Selecting a Specific Function for Two-Pass Inlining

To specify particular functions for consideration for inlining:



Specify the functions to consider for inlining in the **Optimization→Individual Functions→Inline Specific Functions** option (**-OI=fn1[,fn2...]**).

This optimization allows you to specify a list of functions to be considered for inlining. This is similar to manual (two-pass) inlining in that the user determines whether or not each function will be inlined, but does not require modification of the source code. Like **-OI**, **-OI=** does two-pass compilation.

Any functions that have been manually specified (see “Manual Inlining” on page 318) will also be considered for inlining.



Tip You can specify both **-OI** and **-OI=fn[,fn2...]** on the same command line.

Example 2. Two-Pass Inlining: Combining the Inlining Option Variants

Given the following contents of **file_a.c** and **file_b.c**:

```
file_a.c:
int a_very_big_function()
{
    /* lots of code in here, not shown for brevity... */
    return 0;
}
int b()
{
    return 2;
}
```



```
file_b.c:
extern int a_very_big_function(), b();
static inline int c()
{
    return 3;
}
int main()
{
    return a_very_big_function() + b() + c();
}
```

there are several ways to use the inlining option variants.

Given the command line:

```
ccppc -OI file_a.c file_b.c:
```

- The function `a_very_big_function()` will not be inlined, even though **Optimization→Intermodule Inlining (-OI)** is enabled, because the compiler deems it to be too large.
- The function `b()` will be inlined because **Optimization→Intermodule Inlining (-OI)** is enabled and `b()` is small.
- The function `c()` will be inlined because it was declared with `inline` in the same module as the caller.
- The functions `a_very_big_function()` and `b()` will be generated in closed form in module a. The function `c()` will not be generated in closed form because it is declared with `inline` and it was successfully inlined everywhere.

Given the command line:

```
ccppc -OI=a_very_big_function file_a.c file_b.c
```

- The function `a_very_big_function()` will be inlined because it was specified with **Optimization→Individual Functions→Inline Specific Functions (-OI=a_very_big_function)**.
- The function `b()` will not be inlined because **Optimization→Intermodule Inlining (-OI)** was not enabled.
- The function `c()` will be inlined because it was declared with `inline` in the same module as the caller.

- The functions `a_very_big_function()` and `b()` will be generated in closed form in module `a`. The function `c()` will not be generated in closed form.

Given the command line:

```
ccppc file_a.c file_b.c
```

- The functions `a_very_big_function()` and `b()` will not be inlined because **Optimization→Intermodule Inlining (-OI)** was not enabled.
- The function `c()` will be inlined because it was declared with `inline` in the same module as the caller.
- The functions `a_very_big_function()` and `b()` will be generated in closed form in module `a`. `c()` will not be generated in closed form.

Varying Inlining Thresholds

To increase the size threshold beneath which the compiler will consider functions for inlining:



Set the **Optimization→Optimization Scope→Inline Larger Functions** option to **On (-OB)**.

The compiler will still not consider very large functions for inlining unless they are specified using the **Optimization→Individual Functions→Inline Specific Functions** option (**-OI=fn1[,fn2...]**). For example, if **Optimization→Intermodule Inlining (-OI)** is not inlining sufficiently large functions for your needs, pass:

```
ccppc -OI -OB
```

If you need even larger functions to be inlined, then you must specify them individually using **Optimization→Individual Functions→Inline Specific Functions** option (**-OI=fn1[,fn2...]**). For example, to enable automatic two-pass inlining with the higher **Optimization→Optimization Scope→Inline Larger Functions (-OB)** threshold, and to additionally inline the function `a_very_big_function()`, pass:

```
ccppc -OI -OB -OI=a_very_big_function
```

Advantages of Inlining

Inlining is traditionally considered an optimization that improves program speed at the cost of increasing program size. However, although inlining creates more code (because the compiler may generate a single function multiple times within the program), it may actually decrease the overall program size. When a call is replaced by inlined code, the compiler can usually avoid saving and restoring several registers before and after the call. Parameters which normally must be passed on the stack to a called routine can be accessed directly by the inlined routine in their original location. Furthermore, because Green Hills compilers perform inlining before most global optimizations, the process of inlining may significantly enhance the opportunities for additional optimizations. This can result in very efficient code.

For example, if one or more parameter values are constant, large portions of the inlined routine may be reduced or eliminated at compile time, and loops which normally execute a variable number of times may become constant.

Register allocation may improve because inlining eliminates the overhead associated with a call. On most architectures, any function call within a routine reduces the number of registers available for local variables and temporaries. If all function calls can be eliminated by inlining, the number of registers available for variables and temporaries increases.

Pessimistic assumptions made by the compiler when compiling the caller may not be necessary if no call is made. Normally the compiler must assume that when a call is performed, global variables may be changed. This prevents the compiler from optimizing the values of expressions containing global variables across a call to a function. When the function is inlined, the call is eliminated and the global variables can be optimized more freely.



Note Source line number information related to inlined routines is deleted. When a program is executed under the control of a source debugger, no source code is available for the inlined routine. Single stepping by source line will cause the entire inlined call to be executed as a single statement. You can, however, debug the inlined call by stepping through the sequence of inlined machine instructions at the point of the source-level call.

Inlining of C Memory Functions

To disable this optimization:



Set the **Advanced→Optimization Options→Inline C Memory Functions** option to **Off** (-Onomemfuncs).

This optimization enables automatic inlining of calls to C memory functions (if **string.h** is included).

This option may produce multiple copies of string functions; do not use it if those functions require unique addresses. This option's behavior is not strictly standards-compliant.



Note This optimization will not be enabled by default if you set **Optimization→Optimization Strategy to Optimize for Size (-Osize)**. To invoke it in this situation, set the **Advanced→Optimization Options→Inline C Memory Functions** option to **On** (-Omemfuncs).

-Omemfuncs inlines `memmove()`, `memchr()`, and `memcmp()`.

Inlining of C String Functions

To disable this optimization:



Set the **Advanced→Optimization Options→Inline C String Functions** option to **Off** (-Onostrfuncs).

This optimization enables automatic inlining of calls to C string functions (if **string.h** is included).

This option may produce multiple copies of string functions; do not use it if those functions require unique addresses. This option's behavior is not strictly standards-compliant.



Note This optimization will not be enabled by default if you set **Optimization→Optimization Strategy to Optimize for Size (-Osize)**. To invoke it in this situation, set the **Advanced→Optimization Options→Inline C Memory Functions** option to **On** (-Ostrfuncs).

-Ostrfuncs inlines `strlen()`, `strchr()`, `strrchr()`, `strncpy()`, `strcmp()`, `strncmp()`, `strcat()`, `strncat()`, `strstr()`, `strindex()`, and `strrindex()`.

Additional C++ Inlining Information

Although the C++ standard dictates that all member functions defined inside class definitions are implicitly declared inline, the compiler does not consider all such functions for inlining. The reason is that small amounts of code in C++ can expand into dozens of lines in assembly code. If the compiler were to inline every function that is implicitly declared inline, it might create unacceptably bloated code (especially when size is more important than speed). As a consequence, the compiler inlines by default functions composed entirely of straightline code.

If the default C++ inlining behavior does not inline enough functions (the program is too slow) or inlines too many functions (the program is too big), you can override the default behavior as follows:



Set the **Advanced→C/C++ Compiler Options→C++ Inlining Level** option to one of the following settings:

- **Maximum (--max_inlining)** — Instructs the compiler to consider all functions for inlining, whether or not they contain control flow statements, with the exception of excessively large or complicated functions.
- **Maximum Unless Debugging (--max_inlining_unless_debug)** — Behaves the same as **Maximum**, but disables inlining if debugging is enabled. The debugger can only debug inlined functions at the assembly level. The option avoids having to change the inlining options when debugging the application.
- **Standard (--inlining)** — [default] Applies normal C++ inlining rules as described above.
- **Standard Unless Debugging (--inlining_unless_debug)** — Behaves the same as **Standard**, but disables inlining if debugging is enabled.
- **None (--no_inlining)** — Disables all C++ inlining. For more information, see “C++ Inlining Level” on page 303.

You can use these options to control C++ inlining for all modules that make up an executable, or module-by-module. You can also combine them. For example, you could enable maximum inlining for modules that contain only small functions and disable all inlining for modules that contain large functions.

To exclude certain functions from inlining within a C++ file, even if they are member functions defined inside a class, place the `#pragma ghs startnoinline` directive before those functions and the `#pragma ghs`

`endnoinline` directive after. The `__noinline` keyword may be used as a synonym, and has the same effect as wrapping a function declaration with the `startnoinline/endnoinline` pragmas. The pragmas have no effect on template functions or member functions of template classes.

To exclude template member functions or member functions of template classes, use the `__noinline` keyword by adding it to the source code in the same place where you would normally use the `inline` keyword. The `__noinline` keyword treats an inline member function as though it were defined outside of the class body. This has the effect of generating a single, global, out-of-line copy for that template instantiation. This is preferable to disabling all inlining, because it does not generate duplicate copies in different modules.

To make the `__noinline` keyword have this effect:



Set the **C/C++ Compiler**→**C++**→**Keyword Support**→**Support `__noinline` Keyword** option to **On** (`--enable_noinline`).



Note This option is automatically enabled when you set the **Optimization**→**Optimization Strategy** option to **Optimize for Size** (`-Osize`), unless you have enabled `--instantiate_extern_inline`.

Consequently, you can compile the same source code differently without having to remove the `__noinline` keyword. To disable the effect of the `__noinline` keyword, set the **C/C++ Compiler**→**C++**→**Keyword Support**→**Support `__noinline` Keyword** option to **Off** (`--disable_noinline`).

The Green Hills C++ library header files make use of the `__noinline` keyword on some member functions which are known to cause large code bloats.

Interprocedural Optimizations

Interprocedural optimizations allow optimizations based on knowledge of functions being called, such as interprocedural alias analysis. **-Ointerproc** does not require that the entire program is available during compilation (see “Wholeprogram Optimizations” on page 333). Because interprocedural optimizations require analysis of many files at the same time, your host machine may run out of memory if your project is large.

To enable interprocedural optimizations:



Set the **Optimization→Interprocedural Optimizations** option to **Interprocedural (-Ointerproc)**.

Interprocedural options are incompatible with the option **--one_instantiation_per_object**, as well as export templates (enabled with **--export**). Furthermore, interprocedural optimizations are not compatible with the prelinker, and thus require the use of link-once template instantiation.

Related Topics:

- “Link-Once Template Instantiation” on page 637.
- “Exported Templates” on page 641.

This class of optimizations analyzes interaction between functions, and includes:

- **Alias analysis** — Determines what memory can be read and written by function calls. Without this optimization, the compiler assumes that most global memory can be read and written by function calls.

To disable read-based interprocedural alias analysis:



Set the **Optimization→Optimization Scope→IP Alias Reads** option to **Off (-Onoipaliasreads)**.

To disable write-based interprocedural alias analysis:



Set the **Optimization→Optimization Scope→IP Alias Writes** option to **Off** (-Onoipaliaswrites).

To disable interprocedural alias analysis of some standard C functions:



Set the **Optimization→Optimization Scope→IP Alias Lib Functions** option to **Off** (-Onoipaliaslibfuncs).

- **Propagation of constant returns from function calls** — When functions return constant values, those values can be propagated to the caller.

To disable propagation of constant returns from function calls:



Set the **Optimization→Optimization Scope→IP Constant Returns** option to **Off** (-Onoipconstantreturns).

See Example 4 on page 333 for a demonstration on how this option works.

- **Inlining small functions** — Small functions can be inlined across modules.

To disable this inlining of small functions:



Set the **Optimization→Optimization Scope→IP Small Inlining** option to **Off** (-Onoipsmallinlining).

The following example displays a program before and after interprocedural alias analysis is applied.

Example 3. Interprocedural Alias Analysis

Initial C Source Code:

```
int X = 0;
void may_write_X()
{
    printf("Doesn't write anything!\n");
}
void use_global()
{
    X = 1;
    may_write_X();
    if (X != 1)
        do_large_computation();
}
```

Optimized C Source Code:

```
int X = 0;
void may_write_X()
{
    printf("Doesn't write anything!\n");
}
void use_global()
{
    X = 1;
    may_write_X();
}
```

Example 4. Constant Returns

Initial C Source Code

```
int x = 0;
int foo()
{
    x++;
    return 0;
}

void bar()
{
    if (foo())
        x *= 100;
}
```

Optimized C Source Code

```
int x = 0;
int foo()
{
    x++;
    return 0;
}

void bar()
{
    foo();
}
```

Wholeprogram Optimizations

Wholeprogram optimizations analyze program control and data flow at a high level, then optimize your code using several techniques, including:

- Single call-site inlining
- Interprocedural constant propagation
- Interprocedural dead code elimination
- Interprocedural alias analysis

Wholeprogram optimizations can improve program speed and size, often simultaneously.

To enable wholeprogram optimizations:



Set the **Optimization→Interprocedural Optimizations** option to **Wholeprogram (-Owholeprogram)**.

All function call-sites must be made available to the compiler during each compilation. Therefore, if there are functions or variables in the source code that are referenced from files that are being compiled separately or from assembly files and assembly code, you must use the **-external** option to list those functions to keep invalid optimizations from being applied. For more information, see “**Symbols Referenced Externally**” on page 181. A weaker set of intermodule optimizations can be enabled by using **-Ointerproc** (those optimizations are included with **-Owholeprogram**). See “Interprocedural Optimizations” on page 330 for additional information.



Note All symbols used in assembly files, inline assembly, and separate compilation units must be marked as externally visible to ensure that Wholeprogram optimizations are not incorrectly applied.

You can use **-external**, **-external_file**, `#pragma ghs start_externally_visible`, or the GNU attribute `externally_visible` to specify which symbols are externally visible. Symbols referenced in object files or libraries included during compilation are not required to be listed.

The wholeprogram optimization includes all interprocedural optimizations, as well as:

- **Inlining functions that only have one call-site and are otherwise unreferenced** — If there is only one call to a function, the function call is inlined.

To disable one-site inlining:



Set the **Optimization→Optimization Scope→IP One-Site Inlining** option to **Off** (-Onoiponesiteinlining).

This option can be enabled with interprocedural optimizations, although it is not enabled by default. Set the **Optimization→Optimization Scope→IP One-Site Inlining** option to **On** (-Oiponesiteinlining). This option is only effective with -Ointerproc if most call-sites are visible and you have link-time function deletion enabled (-delete).

- **Constant propagation on function parameters** — Conventional constant propagation is performed on the arguments of function calls.

To disable constant propagation on function parameters:



Set the **Optimization→Optimization Scope→IP Constant Propagation** option to **Off** (-Onoipconstprop).

- **Deletion of unneeded and constant function returns** — When the result of a function call is not required, the return statement is removed.

To disable deletion of unneeded and constant function returns:



Set the **Optimization→Optimization Scope→IP Remove Returns** option to **Off** (-Onoipremovereturns).

- **Deletion of unneeded and constant parameters** — When certain parameters are not required, the parameters are removed, and function calls do not pass arguments for them.

To disable deletion of unneeded and constant parameters:



Set the **Optimization→Optimization Scope→IP Remove Parameters** option to **Off** (-Onoipremoveparams).

- **Propagation of constant-value global variables** — When a global variable is initialized to a constant value and never changed, and the variable is not indirectly referenced, the initial value is propagated into the uses of the variable. A limited form of this optimization is also applied to static variables when interprocedural optimizations are enabled.

To disable the propagation of constant-value global variables:



Set the **Optimization→Optimization Scope→IP Constant Globals** option to **Off** (-Onoipconstglobals).

- **Deletion of unreachable functions** — Code is not produced for functions that are unreachable. A limited form of this optimization is also applied to static functions when interprocedural optimizations are enabled. Functions that have all uses inlined may still be removed.

To disable the **-Owholeprogram** deletion of unreachable functions:



Set the **Optimization→Optimization Scope→IP Delete Functions** option to **Off** (-Onoipdeletefunctions).

- **Deletion of constant-value global variables** — Symbols are not produced for global variables that have their constant values propagated to all of their uses. A limited form of this optimization is also applied to static variables when interprocedural optimizations are enabled.

To disable the deletion of constant-value global variables:



Set the **Optimization→Optimization Scope→IP Delete Globals** option to **Off** (-Onoipdeleteglobals).

- To disable all interprocedural and wholeprogram optimizations:



Set the **Optimization→Interprocedural Optimizations** option to **Off** (-Onoipa).

The following example displays a program before and after the interprocedural constant propagation is applied.

Example 5. Interprocedural Constant Propagation

Initial C Source Code

```
int can_be_simple(int x, int y)
{
    if (x != 0 || y > 2)
        do_large_computation();
    return x+y;
}
int is_simple()
{
    return can_be_simple(0, 1) + can_be_simple(0, 2);
}
```

Optimized C Source Code

```
int can_be_simple(int y)
{
    return y;
}
int is_simple()
{
    return can_be_simple(1) + can_be_simple(2);
}
```

Using gbuild with Interprocedural Optimizations

When using Wholeprogram optimizations, you must make all parts of your program available to the compiler for analysis, or the compiler might apply those optimizations incorrectly.

To configure the compiler to perform Wholeprogram analysis on all parts of your program, but prevent the compiler from performing Wholeprogram optimizations on certain parts of that program:

- Set **Optimization→Interprocedural Optimizations** to **Wholeprogram** (**-Owholeprogram**) for your program.
- Set **Optimization→Interprocedural Optimizations** to **Analysis Without Optimizations** (**-Oip_analysis_only**) for the parts of the program that you want to exclude from Wholeprogram optimizations.

When you set **Optimization→Interprocedural Optimizations** to **Interprocedural** (**-Ointerproc**), **-Oip_analysis_only** is not necessary

because the compiler does not need to analyze the entire program to perform optimizations on each part. If you want the compiler to optimize some parts of a program with **-Ointerproc**, but not others:

- Set **-Ointerproc** for your program.
- Set **Optimization→Interprocedural Optimizations to Off (-Onoipa)** for each part of the program on which you do not want to have interprocedural optimizations applied.



Note When the compiler compiles files together, those files cannot use different levels of interprocedural optimization. For example, you cannot enable **-Owholeprogram** on some files and **-Ointerproc** on others. However, you can do so with **-Owholeprogram**, **-Oip_analysis_only**, and **-Onoipa**, or **-Ointerproc** and **-Onoipa** as in the preceding paragraphs.

If a program contains libraries, there are two interprocedural optimization modes that **gbuild** can run in when compiling those libraries:

- When you set **-Ointerproc** for a program, the compiler compiles that program's libraries separately from the rest of the program.
- When you set **-Owholeprogram** for a program, the compiler compiles that program's libraries separately from the rest of the program. **gbuild** also sets **-Oip_analysis_only** for each of those libraries. As a result, the compiler considers information in those libraries when performing Wholeprogram optimizations on the rest of the program, without performing those optimizations on the libraries themselves.

-Owholeprogram is intended for programs only. If you enable it for both a program and an included library, the compiler analyzes and optimizes that library on its own, without regard to the program it is included in.

Loop Optimizations (including SIMD Vectorization)

The compiler uses most of its resources to optimize code in the innermost loops in your source files. Therefore, this optimization is most effective for code containing many loop structures that are executed frequently.

This optimization includes:

- strength reduction
- loop invariant analysis
- loop unrolling

Automatic Loop Optimization



To control this optimization, use the **Advanced→Optimization Options→Optimize All Appropriate Loops** option (**-OL/-Onoloop**).

If code size is a priority, you should set **Advanced→Optimization Options→Loop Unrolling** to **Off** (**-Onounroll**).

Loop Optimizing Specific Functions

To specify particular functions for consideration for loop optimizations:



Specify the functions to consider for loop optimization in the **Optimization→Individual Functions→Loop Optimize Specific Functions** option (**-OL=fn1[,fn2...]**).

For example, the following command line specifies that loop optimizations will be applied to sub and sub2:

```
ccppc -OL=sub,sub2 main.c prog1.c prog2.c
```

Strength Reduction

This optimization is applied to arrays subscripted with the loop index. Most compilers access the array element by multiplying the size of the element by the loop index. The Green Hills compilers store the address of the array in a register and add the size of the array element to the register on each iteration of the loop.

Initial C source code	Optimized C source code
<pre>subr() { int i; int q[4]; for (i=0;i<4;i++) q[i]=i; }</pre>	<pre>subr() { int i; int q[4]; int *_ptr; for (i=0, _ptr=q; i<4; i++) *_ptr++ = i; }</pre>

Strength reduction also applies to multiplying a loop invariant with the loop index. The optimizer replaces a multiply instruction or a call to the **mul()** library function with add and shift instructions.

Loop Invariant Analysis

This optimization examines loops for expressions or address calculations that do not change within the loop. Where such computations are identified, they are relocated to outside the loop and their values are stored in registers.

This optimization is especially useful for reducing code generated to access an array element when the array index does not change within the loop.

Initial C source code	Optimized C source code
<pre>subr() { int i,j; int q[4],p[4]; for (i=3;i>=0;i--) { q[i]=i; for (j=0;j<4;j++) p[j]=q[i]; } }</pre>	<pre>subr() { int i,j; int q[4],p[4]; int tmp; for (i=3; i>=0; i--) { q[i] = i; for (j=0, tmp = q[i]; j<4; j++) p[j] = tmp; } }</pre>

Loop Unrolling

This optimization duplicates the code in innermost loops to produce more straightline code. This removes much of the loop overhead required for testing for stop condition and branching, and allows for more instruction pipelining and better use of the register allocator. It is most effective when the innermost loop is relatively short, causing minimal increase in code size. The following options allow further control over loop unrolling:



To increase the maximum size of loops that may be unrolled:

Set the **Optimization**→**Optimization Scope**→**Unroll Larger Loops** option to **On** (**-Ounrollbig**).

To disable loop unrolling (recommended when code size is a priority):

Set the **Advanced**→**Optimization Options**→**Loop Unrolling** option to **Off** (**-Onounroll**).

The following example uses a loop with a constant iteration count of 100 and an unrolling factor of 4.

Initial C source code	Optimized C source code
<pre>subr(a) int a[]; { int i; for (i=0;i<100;i++) a[i]=i; }</pre>	<pre>subr(a) int a[]; { int i; for (i=0;i<100;i+=4) { a[i]=i; a[i+1]=i+1; a[i+2]=i+2; a[i+3]=i+3; } }</pre>

The initial code has:

- 1 comparison
- 1 memory store
- 1 increment
- 1 branch

for a total of 4 instructions per iteration. Assuming each instruction takes exactly one cycle to complete, and assuming time outside of the loop is negligible, this code takes $100 * 4 = 400$ cycles to complete.

The optimized code has:

- 1 comparison
- 3 additions
- 4 memory stores
- 1 increment
- 1 branch

for a total of 10 instructions per iteration. Using the same assumptions, this code takes $100 / 4 * 10 = 250$ cycles to complete, for a 37.5% improvement.

SIMD Vectorization

SIMD (*Single Instruction Multiple Data*) instructions can perform operations on multiple data items in parallel. Vectorization takes advantage of this capability. It is similar to loop unrolling, except that the compiler generates SIMD code to make the loop's iterations happen in parallel, not serially. The Green Hills compiler uses the AltiVec technology found in AltiVec-enabled processors to generate vectorized code for loops.

You can either use a `#pragma` to manually specify individual loops for vectorization, or use the automatic vectorization mode, which will consider all loops for this optimization.

Manual Vectorization

Manual vectorization is controlled through the following `#pragma` directive:

```
#pragma ghs vectorize[=n]
```

This `#pragma` must be placed directly before the loop to be vectorized. The optional vectorization factor, *n*, specifies the number of iterations that can be performed in parallel, either 2, 4, 8, 16, 32, or 64. If no factor is specified, the compiler will analyze the loop to determine a factor. The compiler will only perform very basic tests for legality; it is your responsibility to ensure that it is safe to perform multiple iterations of the loop in parallel. It is possible to

generate invalid code when specifying a vectorization factor if the different iterations of the loop are not independent.

A loop is manually vectorized only if the target processor is AltiVec-enabled, there is an associated `#pragma` directive and loop optimizations are enabled with **Advanced→Optimization Options→Optimize All Appropriate Loops (-OL)**.

In this example code fragment, the compiler is instructed by `#pragma ghs vectorize` to vectorize the following loop. The compiler will choose a valid vectorization factor.

```
#pragma ghs vectorize
for (i=0; i < 128; i++)
    a[i]=b[i]+c[i];
```

Automatic Vector Optimization

The automatic vector optimization considers all loops for vectorization. It analyzes the code in each loop to determine if vectorization is possible and safe, and then selects a valid vectorization factor. To enable this optimization for AltiVec-enabled targets:



Set the **Optimization→Automatic Vector Optimization** option to **On (-OV)**.

Over-Aligning Data to Aid Vectorization

So that the compiler can safely load and store vectors of data, it is often advisable to over-align variables by using the following `#pragma` directives:

```
#pragma alignvar (n)
```

See “General Pragma Directives” on page 680.

```
#pragma alignref (ptr, n)
```

Informs the compiler that the data pointed to by *ptr* is aligned to at least *n* bytes. This `#pragma` does not affect the alignment of the referent, which should be specified using `#pragma alignvar (n)`.

Example 6. Using the `alignvar` and `alignref` Pragma Directives

In the following example, the array `data1` is aligned to a 16-byte boundary by `#pragma alignvar`, and the variable `x` in `sum_array`, is known to point to 16-byte aligned data because of the associated `#pragma alignref`.

```
#pragma alignvar (16)
int data1[128];

int sum_array(int *x)
{
    int i, sum=0;
    #pragma alignref (x, 16)
    #pragma ghs vectorize=4
    for (i=0; i < 128; i++)
        sum += x[i];
    return sum;
}
```

Because the compiler may generate different code in the presence of the `#pragma alignref` directive, it is important that you use this directive only when it is certain that the data will be properly aligned.



Note To best benefit AltiVec vector operations (which work on 16-byte quantities), you should over-align data to 16-byte boundaries.

Limitations

The vectorization optimization is subject to certain limitations, including:

- Loops containing the following kinds of constructs cannot be vectorized. The automatic optimization ignores such loops, while the manual optimization produces a warning if instructed to vectorize them:
 - Loops containing function calls (except for some specific intrinsic functions)
 - If statements (except for simple constructs that can be replaced by predicated code)
 - Switch statements
- Dependencies within a vector are not permitted. For example, the following loop could not be safely vectorized, because the previous element of `a` is

used when calculating `a[i]`. Since, after vectorization, `a[i-1]` and `a[i]` might be calculated at the same time, the wrong values will be used.

```
#pragma ghs vectorize=4
for (i=0; i < 128; i++){
    a[i] = a[i-1];
}
```



Note The vectorization optimization is not guaranteed to reproduce the sequence of arithmetic operations that would be executed in the non-vector mode

- The vectorization of a reduction (such that calculating the sum of the elements in an array) is done via an associative transformation. First, partial results are calculated, and then the partial results are combined. This can result in different overflow or underflow conditions.
- Because the vector instructions do not support all data types for all arithmetic operations, a sequence of vector instructions might be required to perform an operation that can be accomplished via a single non-vector instruction. Alternatively, the vectorization optimization can use multiply-add vector instructions to perform two arithmetic operations in a single instruction.

Furthermore, the optimization attempts to use the vector instructions that operate on the given data types that might not be available in non-vector mode. For example, a vector addition instruction may operate directly on 8-bit or 16-bit values without needing to first widen the values to 32-bits, perform the operation, and then narrow the 32-bit value back to the original size.

Automatic Vectorization Validity Checking

The automatic optimization uses several tests to determine if a loop is capable of being vectorized. If any of the tests fail, then the loop will not be vectorized.

These tests include:

- Analyze the vector types that would be required to determine that there is at least one vector length that supports all necessary vector operations within the loop.

- Verify that there are no unsupported operations within the loop being vectorized (i.e. `if`, `switch`, or `goto` statements or function calls). For more information, see “Limitations” on page 345.
- Verify that there are no instances of scalar expansion that would need to be stored and then retrieved from an indexed reference.

For example, the following code would not be vectorized because of the need to have multiple instances of `x[i]` within the inner loop:

```
for (i=0; i < 128; i++){
    for (j=0; j < 128; j++){
        x[i] = a[i][j] * b[i][j];
        c[i][j] += x[i];
    }
}
```

- When vectorizing an outer loop, verify that all inner-loop iteration variables are initialized within the loop code being vectorized. If this condition is not true, then the outer loop will not be vectorized.

For example, the following loop will not be vectorized because `k` is not initialized within the vector loop:

```
for (i=0; i < 128; i++){
    for (j=0; j < 4; j++){
        a[i][k] = b[i][k];
        k++;
    }
}
```

- Verify that all indexed references use indices that are either constant within the loop or loops being vectorized or are dependent upon the iteration variables. For example, the following code would not be vectorized due to the reference `a[p[i]]`:

```
for (i=0; i < 128; i++) {
    a[p[i]] = b[i];
}
```

- Using the vector length, verify that there are no data dependencies between two loop iterations where the data value written in one loop iteration would be used in another loop iteration. For example, if the vector length is determined to be 4, elements of this first loop would not be vectorized:

```
for (i=4; i < 128; i++){  
    a[i] = a[i-3];  
}
```

While this second loop would be vectorized:

```
for (i=4; i < 128; i++){  
    a[i] = a[i-4];  
}
```

This test is very conservative and only considers whether the reference spaces overlap, not whether a conflict occurs. For instance, the following two references to `a` are viewed as a potential conflict even though no conflict will occur due to the loop stride. As a result, the following loop will not be vectorized:

```
for (i=0; i < 128; i+=2){  
    a[i] = a[i-1];  
}
```

- The compiler will verify that there are no possible aliases between array or pointer references. If there are potential aliases, then it is possible that the dependency tests will miss possible conflicts and that it is therefore not safe to vectorize the loop. For example, in the following routine it is not possible to determine if there are possible data dependencies because of the potential alias between `a` and `b`. Therefore, this loop would not be vectorized:

```
void sub( int a[], int b[] ) {  
    int i;  
    for ( i=0; i < 128; i++ ) {  
        a[i] = b[i];  
    }  
}
```

However, if the `restrict` keyword is used to indicate that `a` does not have any potential aliases, then this loop would be vectorized:

```
void sub( int a[restrict], int b[] ) {  
    int i;  
    for ( i=0; i < 128; i++ ) {  
        a[i] = b[i];  
    }  
}
```

- Though the primary focus of automatic vectorization is for indexed references, it will support pointer-style code. However, alias checking will frequently prevent valid code from being vectorized. For example, this loop will be vectorized:

```
void sub( int * restrict a, int *b ) {  
    int i;  
    for (i=0; i < 128; i++){  
        *a = *b;  
        a++;  
        b++;  
    }  
}
```

While this loop would not be vectorized due to potential aliases:

```
int *a;  
void sub( int *b ) {  
    int i;  
    int *c;  
    c = a;  
    for (i=0; i < 128; i++){  
        *c = *b;  
        c++;  
        b++;  
    }  
}
```

Loop Peeling

A simple heuristic is used for performing loop peeling prior to vectorizing. The goal of loop peeling is to reduce the number of unaligned vector loads and stores. Given that most loops use array references based on the iteration value, the heuristic peels off iterations from the beginning of the loop to make the iteration value a multiple of the unroll factor. For example, consider the following loop:

```
#pragma alignref(a,8)
#pragma ghs vectorize=4
for( i=1; i<128; ++i)
    a[i] = 0;
```

Since the loop iteration starts at 1, the first reference to `a[i]` is not aligned. In this case, the algorithm will peel off the first 3 loops iterations into a serial loop and then vectorize the loop body starting at iteration `i=4`.



Note Peeling is on by default. To disable this option:



Set the **Advanced**→**Optimization Options**→**Vector Peeling** option to **Off** (`-Onovectorpeel`).

- Peeling is only applied when the increment to the loop iteration variable is positive.
- Peeling is only used for code utilizing indexed array references, code containing pointers being updated will not be peeled.
- When peeling is performed, either two or three loop bodies will be generated, as necessary:
 - The peeled loop body containing serial code
 - The vectorized loop body
 - A final loop body with serial code to complete any remaining iterations (this third loop body will not be generated if the compiler determines, based upon analyzing the loop bounds, that it is not needed).

General Optimizations

To enable the optimizations in this section:



Set the **Optimization→Optimization Strategy** option to one of the following settings:

- **Maximum Debugging and Limited Optimizations (-Omoredebug)**
- **Optimize for Debuggability (-Odebug)**
- **Optimize for Size (-Osize).**
- **Optimize for General Use (-Ogeneral).**
- **Optimize for Speed (-Ospeed).**

The following options have no effect without an optimization strategy selected. They are all enabled when a strategy is selected, except when noted otherwise.

Peephole Optimization

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see “General Optimizations” on page 351). To control it individually:



Use the **Advanced→Optimization Options→Peephole Optimization** option (-Opeep/-Onopeephole).

You may find it helpful to select the **Restrict peephole optimization scope to increase debuggability** setting of the **Optimization→Optimization Scope→Pipeline & Peephole Scope** option (-Olimit=peephole).

This optimization identifies common code patterns and replaces them with more efficient code. This includes optimizations such as flow of control, algebraic simplifications, and removal of unreachable code. The compiler only performs this optimization when local code analysis ensures that the results will be correct without further analysis of the surrounding code.

Initial pseudo code	Optimized pseudo code
<pre>mul r1 <- r2, r3 add r4 <- r1, r4</pre>	<pre>mac r4 <- r2, r3</pre>

Pipeline Instruction Scheduling

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see “General Optimizations” on page 351). To control it individually:



Use the **Advanced→Optimization Options→Pipeline Instruction Scheduling** option (**-Opipeline/-Onopipeline**).

When debugging, you may find it helpful to select the **Restrict pipeline optimization scope to increase debuggability** setting of the **Optimization→Optimization Scope→Pipeline & Peephole Scope** option (**-Olimit=pipeline**).

This is a low-level time optimization from which a majority of architectures benefit. These architectures (which include all RISC implementations, most new CISC implementations, many DSP processors, and even new application-specific cores) are pipelined, and usually expose at least certain aspects of that pipeline to users.

Instruction scheduling involves reordering instructions for these architectures to make more efficient use of the associated pipelines. The optimizer uses three methods for pipeline instruction scheduling:

- branch scheduling
- basic block scheduling
- superscalar scheduling (for certain architecture implementations only)

These are among the most important optimizations for many architectures, and can increase code speed for the majority of applications.

Example 7. Branch Scheduling and Basic Block Scheduling

For this generic machine example, assume that:

- there is one delay slot after a branch
- at least one instruction is required between a load into a register and the use of that register
- the last register of a three operand instruction is the destination

	(a) Before Scheduling Pseudo Code	(b) After Scheduling Pseudo Code
1	load r0, 0(r5)	load r0, 0(r5)
2	load r1, 4(r6)	load r1, 4(r6)
3	add r0, r1, r2	add r7, r8, r9
4	add r7, r8, r9	add r0, r1, r2
5	sub r0, r3, r4	branch L2
6	branch L2	sub r0, r3, r4
7	nop	

Branch scheduling consists of two different areas: filling branch delay slots and filling a delay between a comparison and a branch based on that comparison.

In this example, the branch delay slot is filled when instruction 5 from sequence (a) is placed in the `nop` slot after the branch at instruction 6 in sequence (b). This causes the normally unused cycles after the branch to be used on the `sub` pseudo instruction.

Certain architectures require a number of cycles to elapse between a comparison and branch based on that comparison. The method shown above is used to move one or more instructions between the compare and the branch to make use of those cycles.

Basic block scheduling is performed within a basic block (a sequence of instructions which always executes as a group). The instructions within a single basic block are sorted so that the new sequence produces the same result as the original sequence.

In this example, this change is made by reversing the order of the two `add` instructions at lines 3 and 4 between sequence (a) and sequence (b) to allow for the load delay in instruction 2. This will prevent a stall after the load and instead use those cycles on the `add` instruction.

Example 8. Superscalar Scheduling

Scheduling for superscalar implementations involves taking into account the existence of multiple execution units. Many architectures are now designed with multiple integer, floating-point, or load-store units in a single implementation. The compiler can schedule the instructions of these architectures so that the

instructions associated with multiple execution units can execute at the same time.

For this generic superscalar machine example, assume the same architecture characteristics as in the previous example and, also, that:

- one integer and one floating-point instruction can be issued in the same cycle
- the floating-point instruction must follow the integer instruction with which it is paired

This example uses the same instructions as the last example, with the addition of a floating-point add instruction and a floating-point sub instruction at lines 6 and 7.

	(a) Before Scheduling Pseudo Code	(b) After Scheduling Pseudo Code
1	load r0, 0(r5)	load r0, 0(r5)
2	load r1, 4(r6)	load r1, 4(r6)
3	add r0, r1, r2	add r7, r8, r9
4	add r7, r8, r9	addd f0, f1, f2
5	sub r0, r3, r4	add r0, r1, r2
6	addd f0, f1, f2	subd f3, f2, f4
7	subd f3, f2, f4	branch L2
8	branch L2	sub r0, r3, r4
9	nop	

In this example, the two floating-point instructions at lines 6 and 7 from sequence (a) have been moved to lines 4 and 6 in sequence (b), making the most efficient use of the multiple instruction issue capability of the integer and floating-point execution units.

Common Subexpression Elimination/Partial Redundancy Elimination

This optimization is generally controlled by the setting you specify in the **Optimization**→**Optimization Strategy** option (see “General Optimizations” on page 351). It is not enabled by **-Odebug** or **-Omoredebug**. To control it individually:



Use the **Advanced**→**Optimization Options**→**Common Subexpression Elimination** option (**-Ocse/-Onocse**).

This optimization is implemented when the compiler determines that a previously calculated expression is part of a later expression and none of the variable values in the subexpression have changed. The compiler retains the value of the subexpression in a register for reuse.

Initial C source code	Optimized C source code
<pre>int subr(int x, int y) { int a, b; x += a+b; y += a+b; if (y < 0) return (y); return (x); }</pre>	<pre>int subr(int x, int y) { int a, b, _v6; x+=(_v6=a+b); y+=_v6; if (y<0) return (y); return (x); }</pre>

Partial redundancy elimination is similar to common subexpression elimination, except that code may be moved in order to make partially redundant expressions fully redundant. The compiler will choose between these two optimizations as appropriate.

Tail Calls

Tail call optimization is generally controlled by the **Optimization**→**Optimization Strategy** option (see “General Optimizations” on page 351). It is not enabled by **-Odebug** or **-Omoredebug**. To control it manually:



Use the **Advanced**→**Optimization Options**→**Tail Calls** option (**-Otailrecursion/-Onotailrecursion**).

A function `foo()` is considered for tail call optimization if the last executed statement returns the value of a function call `bar()`. With the optimization, the compiler branches from `foo()` to `bar()`, and `bar()` returns directly to `foo()`'s caller. For example:

C source code	Initial assembly	Optimized assembly
<pre>int bar(void); int foo(void) { return bar(); }</pre>	<pre>call bar() restore registers restore stack frame return to foo() restore registers restore stack frame return to caller()</pre>	<pre>restore registers restore stack frame branch to bar() return to caller()</pre>

This option also controls tail recursion optimizations. A function is considered tail recursive if the last statement executed is a call to itself followed by a return statement. This optimization replaces the call with a branch instruction and eliminates the return statement. For example:

Initial C source code	Optimized C source code
<pre>int sum(int n) { if (n <= 1) return (1); else return (n+ sum(n-1)); }</pre>	<pre>int sum(int n) { int _v3=0; L1: if (n <= 1) return _v3+1; _v3 += n; _n--; goto L1; }</pre>

Constant Propagation

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option (see “General Optimizations” on page 351). It is not enabled by **-Odebug** or **-Omoredebug**. To control it manually:



Use the **Advanced→Optimization Options→Constant Propagation** option (**-Oconstprop/-Onoconstprop**).

This optimization replaces a variable with a constant, where the compiler determines that the value of the variable does not change.

Initial C source code	Optimized C source code
<pre>int main() { int i,a,b; a = 3; b = 0; for (i=0;i<1000;i++) { b += a; /* a is constant over the lifetime of the loop */ } printf("%d\n", b); return 0; }</pre>	<pre>int main() { int i,b; b = 0; for (i=0; i<1000; i++) b+=3; printf("%d\n", b); return 0; }</pre>

C/C++ Minimum/Maximum Optimization

This optimization is generally controlled by the setting you specify in the **Optimization**→**Optimization Strategy** option (see “General Optimizations” on page 351). To control it individually:



Use the **Advanced**→**Optimization Options**→**C/C++ Minimum/Maximum Optimization** option (**-Ominmax/-Onominmax**).

This optimization can generate special code for minimum, maximum, and absolute value expressions of the form:

```
(i < j) ? i : j
(i > j) ? i : j
(i < 0) ? -i : i
```

Memory Optimization

This optimization is generally controlled by the setting you specify in the **Optimization**→**Optimization Strategy** option (see “General Optimizations” on page 351). To control it individually:



Use the **Advanced**→**Optimization Options**→**Memory Optimization** option (**-OM/-Onomemory**).



Note This optimization assumes that non-volatile memory locations only change with explicit store instructions and thus are not affected by any external sources. Hence, it is not recommended for applications such as device drivers, and operating systems, in which memory could be externally affected, or for shared memory or a non-virtual memory environment when interrupts are enabled.

This optimization instructs the compiler to optimize repeated memory reads by placing the value in a local temporary location. Subsequent read operations then refer to the register rather than the actual memory location.



Note This optimization is only enabled through the **Optimization→Optimization Strategy** option for ANSI C and C++. To enable it when compiling K&R C code, you must set the **Advanced→Optimization Options→Memory Optimization** option to **On (-OM)**.

To disable memory optimization in ANSI C or C++, use the `volatile` keyword to explicitly declare any object that may change without the compiler's knowledge or control (see "Type Qualifiers" on page 588), or set the **Advanced→Optimization Options→Memory Optimization** option to **Off (-Onomemory)**.

Dead Code Elimination

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option. It is not enabled by **-Odebug** or **-Omoredebug**, and it cannot be controlled individually. It removes code that computes values which are never used and eliminates blocks that are determined to be unreachable.

In the following example, after the expression `0*x` is constant folded and the value `0` is constant propagated into the `if` conditional, this optimization is able to determine that the `then` block is unreachable and removes the call `foo()` and the `return c`. Then, it can determine that the value of variable `c` is no longer used, and is able to eliminate the expression `c = 3 * x`.

Initial C source code	Optimized C source code
<pre>int subr(int x) { int a, b, c; a = 0 * x; b = 2 * x; c = 3 * x; if (a) { foo(); return c; } else { bar(); return b; } }</pre>	<pre>int subr(int x) { int b; b = 2 * x; bar(); return b; }</pre>

Static Address Elimination

This optimization is generally controlled by the setting you specify in the **Optimization→Optimization Strategy** option. It is not enabled by **-Odebug** or **-Omoredebug**, and it cannot be controlled individually. It instructs the optimizer to assign frequently used static variables to registers within the scope of the function. This eliminates the loads and stores required with memory allocation.

In these examples, the address of the static variable `x` is maintained in a register.

Initial C source code	Optimized C source code
<pre>int subr(int q) { static int x=0; x++; q+=x; return(q); }</pre>	<pre>int subr(int q) { static int x=0; register int x_ = x; x_++; q+=x_; x=x_; return(q); }</pre>



Note This optimization is performed not only for locally defined static variables, but also for global variables, as shown in the following example.

Example 9. Using Global Variables with Static Address Elimination

Initial C source code	Optimized C source code
<pre>int x = 0; int subr(int q) { x++; q+=x; return q; }</pre>	<pre>int x = 0; int subr(int q) { register int x_ = x; x_++; q+=x_; x=x_; return(q); }</pre>

Default Optimizations

The optimizations in this section are very basic and are enabled even if no **Optimization**→**Optimization Strategy** is specified.

Register Allocation by Coloring

This optimization is enabled by default. To disable it:



Set the **Advanced**→**Optimization Options**→**Register Allocation by Coloring** option to **Off (-nooverload)**.

This optimization permanently maintains a selected set of local scalar variables in registers, based on their frequency of reference and their lifetimes. The compiler uses data flow analysis to determine the lifetime of each variable. The register allocator uses this information to assign different variables within a function to the same register if the lifetimes of the variables do not overlap. This increases the opportunity for allocating variables to registers, which in turn enables the compiler to significantly optimize the code, (since additional memory load and store instructions are not required to reference the variables).

In the following example, the variables a and b are both assigned to the same register because their lifetimes do not overlap (note that the code could be optimized further, but is not here in order to simplify the example).

Initial C source code	Optimized C source code
<pre>int subr(int x) { int a,b; a=x; b=x*2; return b; }</pre>	<pre>int subr(int x) { int a; a=x; a=x*2; return a; }</pre>

Scalar variables are generally considered for register allocation unless their values are accessed with the address operator (&).

Automatic Register Allocation

This optimization is enabled by default. To disable it:



Set the **Advanced**→**Optimization Options**→**Automatic Register Allocation** option to **Off** (-noautoregister).

This optimization enables the automatic allocation of local variables to registers.

Initial C source code	Optimized C source code
<pre>extern int global; int foo(int e) { int local = e; if (global) local = local * 2; return local; }</pre>	<pre>extern int global; int foo(int e) { register int local = e; if (global) local = local * 2; return local; }</pre>

Register Coalescing

This optimization is enabled by default and cannot be disabled. It eliminates the additional register-to-register copies required when using a temporary register. When evaluating an expression, the compiler uses the destination register as a work register and organizes the instruction sequence so that the result ends up in that register.

Initial C source code	Optimized C source code
<pre>int fun(int a,int b,int c) { int ret = a+b+c; return ret; }</pre>	<pre>int fun(int a,int b,int c) { return a+b+c; }</pre>

Constant Folding

This optimization is enabled by default, and cannot be disabled. If the compiler determines that an expression is a constant, it substitutes the constant in place of any reference to the expression. In the following example, the expression `SHRT_MAX/2` is a constant with a value of 16383.

Initial C source code	Optimized C source code
<pre>#define SHRT_MAX 32767 short subr() { int x; x=SHRT_MAX/2; return(x); }</pre>	<pre>short subr() { int x; x = 16383; return(x); }</pre>

Loop Rotation

This optimization is enabled by default and cannot be disabled. It locates the termination test and a conditional branch at the bottom of the loop. This causes the loop to process only one branch instruction on each iteration.

Initial C source code	Optimized C source code
<pre>int subr(int i) { while (i < 10) i *= i; return(i); }</pre>	<pre>int subr(int i) { goto L7; do { i *= i; L7: } while (i < 10); return(i); }</pre>

If the compiler determines that the loop is executed at least once, it is entered at the top. If not, the compiler generates an unconditional branch before the loop to the termination test.

The DoubleCheck Source Analysis Tool

Chapter 8

This Chapter Contains:

- Introduction
- Enabling DoubleCheck
- Using Custom Functions with DoubleCheck
- Viewing DoubleCheck Reports
- Controlling DoubleCheck Output
- Using DoubleCheck With Run-Time Error Checking
- Source Analysis Error and Warning Messages

Introduction

DoubleCheck is a tool for identifying bugs in C and C++ code. It finds code that, when executed, might cause unexpected behavior or harmful corruption of other programs. While a typical compiler alerts you to basic potential code problems, DoubleCheck is a powerful static analysis tool that can analyze large pieces of code spanning many source files and find bugs caused by complex interactions between these pieces of code. Without DoubleCheck, you must track down bugs with intensive debugging after they manifest themselves in a running program. By using DoubleCheck, you can catch these bugs without even running the program.

DoubleCheck works by finding potential execution paths through code, and then determining how the values of variables can change over these paths. The variables can be in memory or registers. The execution paths can proceed through function calls in the same or different files.

DoubleCheck runs at the same time a program is compiled so that no separate tool or separate pass is required. After you enable DoubleCheck in your project, any time you build all or part of the project, DoubleCheck quickly analyzes code and issues errors and warnings illustrating how the bugs it finds might occur.

DoubleCheck can also produce a report file containing extensive information about the bugs it identifies. The report explains where and under what circumstances the bugs exist. It also provides statistics about the project, such as the number of lines of source code, and statistics about the bugs it finds, such as the number of errors per line of source code. DoubleCheck breaks the statistics down by source file and directory. You can view the report using the provided report viewer (see “Enabling DoubleCheck” on page 367), which allows you to sort and mask different error types for a custom view that fits your needs.

DoubleCheck is pre-configured with knowledge of properties of standard library functions. It uses this information to detect errors in code that calls these functions. For example, DoubleCheck checks that when calling the function `free()`, the caller only passes a pointer to memory allocated by functions like `malloc()`.

You can also assign these properties to user-defined functions. For example, if you use a custom memory allocation system, you can assign properties to functions in that system to aid DoubleCheck in looking for memory allocation bugs. By assigning properties to functions, you also reduce the number of *false*

positives, which are potential bugs identified by DoubleCheck that do not cause errors during program execution.

Enabling DoubleCheck

When you enable DoubleCheck, the compiler outputs source analysis errors and source analysis warnings along with the normal compiler errors and warnings as it builds source files. To enable DoubleCheck, use the **-double_check.level=setting** option, where *setting* is one of the following:

- **None (-double_check.level=none)** — [default] DoubleCheck is disabled.
- **Low (-double_check.level=low)** — Looks for all types of bugs, but does not follow execution paths involving function calls. DoubleCheck runs quickly and should not affect the time required to compile files.
- **Medium (-double_check.level=medium)** — Looks for all types of bugs, and follows execution paths through calls between functions in all source files. This extra processing requires more time than the low level.
- **High (-double_check.level=high)** — Extends the medium level by investigating even more inter-function execution paths. This requires more processing time than the medium level.

Each level performs a different degree of processing. At a higher level, DoubleCheck can identify more bugs than at a lower level, but will require more time to analyze your code.

The medium and high levels require a two-pass compilation. The first pass summarizes the contents of source files. The second pass uses the summaries of all source files when compiling each file, which identifies potential intermodule paths.

Adjusting compiler inlining settings may affect DoubleCheck analysis. For example, enabling inlining may provide DoubleCheck with more execution pathways to analyze, while disabling inlining may provide DoubleCheck with more out-of-line function copies to analyze.

Specifying a DoubleCheck Report File

The most useful way to view DoubleCheck output is to generate a report file and open it using the Web-based report viewer. Report files contain more

detailed information than the errors and warnings DoubleCheck issues as source files are built.

To set the name of the report file DoubleCheck generates, use the **-double_check.report=Report_File.gsr** option, where *Report_File.gsr* is the name of the file. For example:

```
-double_check.report=Project1.gsr
```

After you have specified a report file and built your application, you can view the report by right-clicking a project in the Project Manager and selecting **View DoubleCheck Report**. See “Viewing DoubleCheck Reports” on page 373 for additional report viewing methods.



Note The report contains information about the most recent build of source files. If you fix an error and rebuild the corresponding source file, the report will no longer show the error. When working with projects that cause a source file to be built multiple times, we recommend that you generate multiple report files in order to maintain errors from each build.

Using Custom Functions with DoubleCheck

DoubleCheck uses pre-configured properties of standard C and C++ library functions to determine how it should analyze calls to those functions, and when to report an error or warning. When running at the medium or high level, DoubleCheck can sometimes infer these properties for other functions. When DoubleCheck is running at the low level, or when functions exist in an external library, it is much less likely that DoubleCheck will be able to infer these properties. In these cases, you can tell DoubleCheck about the properties of other functions to increase the number of relevant bugs it finds in your source code. At a minimum, you should specify properties of functions that do not return, such as error handling functions, if they abort execution.

The following sections describe how to specify properties of functions to DoubleCheck, and what those properties mean:

- “Specifying Function Properties in a DoubleCheck Configuration File” on page 369
- “Specifying Function Properties with Pragma Directives” on page 370
- “Property Types” on page 370

Specifying Function Properties in a DoubleCheck Configuration File

You can specify function properties to DoubleCheck by using a configuration file. You do not have to modify source code when using a configuration file. Configuration files specify one property per line, except lines that start with #, which are ignored. The format of each line is:

Property_Type *Argument_Number* *Function_Name*

where:

- *Property_Type* — The property type (for example, `no_return`). See “Property Types” on page 370 for a list of property types.
- *Argument_Number* — The number of the argument to which this property refers. Specify 0 if this property refers to the return value of the function (for example, `needs_null_check`) or to the function itself (for example, `no_return`).
- *Function_Name* — The name of the function to which you are assigning the property.

When using C++, do not specify class or argument types. For example, specify `MyFunc` instead of `MyClass::MyFunc(int my_var)`. This will match all functions named `MyFunc`. Use mangled function names to differentiate the parent class or argument types. You can find mangled function names by running **gdump -nx** or **gnm** on the compiled object module (see Chapter 13, “Utility Programs”).

When naming a DoubleCheck configuration file, use the **.dcc** extension. Specify the configuration file to the compiler with the **-double_check.config=File_Name.dcc** option, where *File_Name.dcc* is the configuration file.

Example 1. DoubleCheck Configuration File

```
# This file defines properties of a function called
# another_malloc to be exactly the same as the standard malloc
needs_null_check 0 another_malloc
malloced 0 another_malloc
malloc_size 1 another_malloc
```

Specifying Function Properties with Pragma Directives

You can also specify function properties to DoubleCheck by inserting `#pragma` directives in your source code. Insert the `#pragma` directive near the declaration of a function in a header file. We recommend this because you are likely to remember to update the `#pragma` directive if you change the function declaration. When you place a `#pragma` directive in a header file, DoubleCheck reads the directive whenever another file `#includes` that header file. The format of the DoubleCheck `#pragma` directives is as follows:

```
#pragma double_check Property_Type Argument_Number Function_Name
```

where *Property_Type*, *Argument_Number*, and *Function_Name* have the same meaning as in “Specifying Function Properties in a DoubleCheck Configuration File” on page 369.

Property Types

The following sections describe the property types that can be used in *Property_Type* (see “Specifying Function Properties in a DoubleCheck Configuration File” on page 369). Some of these property types (such as `needs_null_check`) help DoubleCheck find more bugs, while others (such as `no_return`) help avoid false positives by preventing DoubleCheck from analyzing infeasible execution paths.

needs_null_check

Indicates that a function’s return value might be `NULL`. A program must verify that the function’s return value is not equal to `NULL` before it dereferences that value. The argument number of this property must be 0 because it refers to the function’s return value.

malloced

Indicates that a function’s return value is allocated and must be deallocated before being overwritten or falling out of scope to avoid resource leaks. The argument number of this property must be 0 because it refers to the function’s return value.

If a function with the `malloced` property allocates a resource, that resource should be deallocated by a function with the `frees` property. Use the `malloc_size` property if arguments to this function indicate the size of a memory region that this function allocates.

malloc_size

Indicates that a particular argument specifies the size of a memory region allocated and returned by this function. You can use `malloc_size` twice if the two arguments are multiplied together to specify the size. For example, a function that behaves like `calloc` might use the following properties:

```
malloc_size 1 another_calloc
malloc_size 2 another_calloc
```

frees

Indicates that the specified argument contains a pointer that the function deallocates. If a function with the `frees` property deallocates a resource, that resource should have been allocated by a function with the `malloced` property.

derefed

Indicates that a function argument is unconditionally dereferenced. This is true only if the dereference always happens and is not preceded by a `NULL` check. For example, this `#pragma` directive specifies the correct property for the `df` function:

```
#pragma double_check derefed 1 df
int df(int *x)
{
    return *x;
}
```

no_return_when_zero

Indicates that the function never returns when the specified argument has a value of zero. For example, this `#pragma` specifies the correct property for the `my_check` function:

```
#pragma double_check no_return_when_zero 1 my_check
void my_check(int *x)
{
    if (x == 0) {
        printf("An error has occurred\n");
        exit(1);
    }
}
```

no_return_when_non_zero

Indicates that the function never returns when the specified argument has a non-zero value. For example, this #pragma specifies the correct property for the my_check2 function:

```
#pragma double_check no_return_when_non_zero 1 my_check2
void my_check2(int *x)
{
    if (x) {
        printf("An error has occurred\n");
        while(1);
    }
}
```

no_return

Indicates that a function never returns. The function should always enter an infinite loop or call another no_return function such as exit or abort. The argument number of this property must be 0 because it refers to the function itself.

unsaved

Indicates that a particular argument is never copied to global memory, or saved to any other storage accessible outside a particular call to the function. The function can dereference the argument's value, but should not deallocate the memory region pointed to by the argument. unsaved is useful for helping DoubleCheck catch memory leaks. For example, DoubleCheck can catch the following memory leak even when you run it at the **low** level:

```
#pragma double_check unsaved 1 store
void store(int *x, int y)
{
    *x = y;
}
void buggy()
{
    int *x = (int*)malloc(sizeof(int));
    store(x, 5);        // store does not save x anywhere
}                       // x leaked!
```

Viewing DoubleCheck Reports

You can configure DoubleCheck to output a report file containing information about source analysis errors and warnings. The report contains more detailed information than the errors and warnings DoubleCheck issues as source files are built. The report file also contains some useful metrics and statistics. Viewing the report files with the viewer is the best way to browse DoubleCheck's results.

Launching the Report Viewer from the MULTI Project Manager

To launch the report viewer from the Project Manager, select **Tools** → **View DoubleCheck Report** or right-click a project and select **View DoubleCheck Report**. This menu item is only enabled if the project generates a report (that is, if the `-double_check.level=setting` and `-double_check.report=Report_File.gsr` options are set) and has been built.

Launching the Report Viewer From the Command Prompt

The report viewer is a Web server that you execute using the **gsar** command. You can invoke **gsar** on any DoubleCheck report file. For example:

```
gsar Project1.gsr
```

On Windows hosts, **gsar** launches the default Web browser to display the report. On other hosts, you must use the `-browser=` option to specify a browser to launch. For example:

```
gsar -browser=firefox Project1.gsr
```

gsar Options

Use the following options with **gsar** to control report viewing:

- **-browser=*B*** — Launches Web browser *B* on the main report after starting the Web server. On Windows hosts, **gsar** launches the default Web browser if you do not specify this option.
- **-port=*N*** — Starts the Web server using port *N*. The default port is 13065.
- **-tabsize=*N*** — Substitutes each tab character in your source files with *N* spaces. The default size is 8.
- **-quiet** — Stops **gsar** from printing any status messages.

Controlling DoubleCheck Output

By default, the source analysis errors and warnings generated by DoubleCheck during a build do not interrupt or terminate the build like other compiler-generated errors. To get the most benefit out of DoubleCheck, we recommend that you fix all the bugs that DoubleCheck finds, and then raise the priority of source analysis errors and warnings to the same level as normal compiler-generated errors. Raising the priority prevents future code changes from adding bugs, because the code will fail to build.

Promoting Source Analysis Errors and Warnings

To raise source analysis error and/or warning message priority to stop the build, use **-double_check.stop_build=*setting*** option, where *setting* is one of the following:

- **Off (-double_check.stop_build=off)** — [default] Source analysis error and warning messages do not stop the build.
- **Errors Only (-double_check.stop_build=errors)** — Only source analysis error messages stop the build.
- **Warnings and Errors (-double_check.stop_build=warnings)** — Source analysis error and warning messages stop the build.

Ignoring Source Analysis Errors and Warnings

In certain cases it might be useful to ignore source analysis error and warning messages. For example, if certain parts of your project cannot be modified, you can ignore source analysis errors that occur in those parts. You can ignore source analysis errors and warnings with a build option or by inserting `#pragma` directives in your code.



Note You might also want to disable these messages if they report code paths that you know are impossible. However, if the code is complex enough to fool DoubleCheck, it is probably complex enough to fool a less experienced programmer into making a change that introduces a bug. Instead of ignoring DoubleCheck, we recommend that you change the code's control flow to avoid the possibility of uncovering problems later.

Using a Builder Option to Ignore Source Analysis Errors and Warnings

To disable a particular source analysis error or warning message for a file or part of a build hierarchy, use the `-double_check.ignore=number` option, where *number* is the number of the error or warning you want to ignore. For example, to ignore this error:

```
source analysis error #5: potentially NULL pointer "x"
      dereferenced
"/temp/file038.c", line 24: pointer set to NULL
    int i, *x = 0;
           ^
"/temp/file038.c", line 33: pointer dereferenced
    *x++ = 4;
      ^
```

use the `-double_check.ignore=5` option.

Using `#pragma` Directives to Ignore Source Analysis Errors and Warnings

To ignore a particular source analysis error or warning on one line of code, place the following `#pragma` before that line:

```
#pragma double_check ignore number
```

where *number* is the number of the error or warning that you want to ignore, or an asterisk (*) if you want to disable all types of errors and warnings.

To ignore a particular source analysis error or warning on several lines of code, place the following `#pragmas` before and after the lines of code:

```
#pragma double_check start_ignore number
/* your code with ignored errors goes here */
#pragma double_check end_ignore number
```

Using DoubleCheck With Run-Time Error Checking

Enabling run-time error checks may cause DoubleCheck to miss potential bugs that it would otherwise report. To use both DoubleCheck and run-time error checking to debug your program, perform two separate builds, where one build uses DoubleCheck and the other uses run-time error checking.

For more information about run-time error checking, see “**Run-Time Error Checks**” on page 188.

Source Analysis Error and Warning Messages

DoubleCheck can identify many different types of bugs. The following list explains why you should fix these bugs, how you can fix them, and coding practices you can use to avoid them in the future.

Error Messages

1: Pointer dereferenced before first being NULL-checked

DoubleCheck issues this error when your program calls a function that returns a pointer that is potentially NULL, and then dereferences the resulting pointer without performing a *NULL-check*. A NULL-check is an `if` statement that compares the value of a pointer to NULL to determine whether or not it is a NULL pointer.

DoubleCheck knows which standard library functions potentially return NULL. For other functions, DoubleCheck looks to see if you have assigned the `needs_null_check` property to that function.

Importance

Unexpected results might occur when a program dereferences a NULL pointer. The dereference might access the value at memory address 0. The dereference might trigger an exception. If you do not check the pointer before you dereference it, you might lose the ability to properly handle a memory allocation failure (for example, by producing a failure message indicating where in the code the failure occurred and how much memory was requested). Knowing where in code the problem occurred makes it easier to track down the cause of memory exhaustion. Knowing how much memory was requested can help you determine if a negative size value was accidentally passed to a function such as `malloc`.

Avoidance

Use a simple function to check the return value of library functions like `malloc`. For example:

mm.h

```
#pragma double_check malloced 0 my_malloc
#pragma double_check malloc_size 1 my_malloc
void *my_malloc(size_t size);
```

mm.c

```
#include <stdlib.h>
#include <stdio.h>
#include "mm.h"

void *my_malloc(size_t size)
{
    void *p = malloc(size);
    if (p == NULL) {
        printf("Allocation of %lu bytes failed.\n", (unsigned long)size);
        exit(1);
    }
    return p;
}
```

2 / 3: Access extends beyond end of / Access extends beyond beginning of

DoubleCheck issues this error when your program adjusts a pointer to a variable or allocated memory region so that it points before the beginning or beyond the end of the region, and then dereferences that pointer.

Importance

Reading past the beginning or end of a variable or allocated memory region returns undefined results. Writing past the end or beginning of an allocated memory region will often corrupt data structures internal to `malloc` and potentially other allocated memory regions. This corruption can cause `malloc` to fail in the future. These bugs can be hard to track down because they can occur long after the write that caused the corruption.

Avoidance

We recommend that you define the size of an array or allocated memory region in only one place in one header file. This enables you to reference one definition in all the source files. Viewing one definition eliminates the chance that someone will change only the part of the code that makes the definition of an array and not how it is accessed elsewhere. We suggest that you do not write code that contains constant sizes:

```
for (index = 0; index < 8; index++)  
    array[index] = 0;
```

Instead, find the bounds of an array using the following technique:

```
#define DIM(a) (sizeof(a)/sizeof((a)[0]))  
for (index = 0; index < DIM(array); index++)  
    array[index] = 0;
```

4: Resource leak through pointer

DoubleCheck issues this error when an execution path exists where a resource is allocated and is not freed, and the pointer to this resource is never saved so it cannot be freed later.

Importance

Over time, resource leaks can lead to resource exhaustion. These bugs can be hard to reproduce and track down because they might only happen when a program is run for a long time or only in a certain way.

Avoidance

To avoid resource leaks, use statically allocated resources instead of dynamic ones. For example, use a global-scoped or static function-scoped variable instead of allocating memory resources with calls to `malloc()`. Statically allocating resources also gives you the benefit of knowing the resource requirements of your application before running it.

If a resource must be dynamically allocated, try creating a clear execution path from the allocation to the point where the resource is freed. If you cannot create a clear execution path, use the C++ `auto_ptr` template. In the following example, `obj` is always freed and never leaked after allocation:

```
#include <memory>
...
void f()
{
    std::auto_ptr<Object> obj(new Object);

    if (error1)
        return;                // obj freed here
    if (error2)
        return;                // obj freed here
}
```

5: Potentially NULL pointer dereferenced

DoubleCheck issues this error when a feasible execution path exists between setting a pointer to NULL and dereferencing that pointer.

Many compilers do not give errors for obvious cases, such as:

```
int *x = NULL;
*x = 0;
```

DoubleCheck catches these cases, as well as complex cases found through very long execution paths with complicated control flow.

DoubleCheck does not consider an execution path to be feasible if control flow checks guarantee that the program will modify the pointer's value after it is set to NULL, but before it is dereferenced. For example, in the following code it is known that no feasible path from POINT A to POINT B exists, and so DoubleCheck will not issue this error:

```
#pragma double_check no_return 0 internal_error

void foo(int x)
{
    int a, b;
    int *y = NULL;  // POINT A

    switch (x) {
        case 0:
            y = &a;
            break;
        case 1:
            y = &b;
            break;
        default:
            internal_error();
            // #pragma indicates we cannot reach this statement
            break;
    }

    *y = 4;          // POINT B
}
```

Importance

Various problems might arise when you dereference a NULL pointer. See “1: Pointer dereferenced before first being NULL-checked” on page 376 to learn more about these problems.

Avoidance

Avoid any potential control flow path between setting a pointer to NULL and dereferencing that pointer. Initializing pointers to NULL avoids the use of uninitialized values. If there is complicated control flow between initializing a pointer to NULL and dereferencing that pointer, perform a NULL-check before dereferencing it:

```
int *x = NULL;
...
// complicated logic that might initialize x
...
if (x != NULL) {
    ...
    // dereference of x
    ...
}
```

6: Access into deallocated memory

DoubleCheck issues this error when your program accesses a memory region after deallocating it. For example, it is unsafe to free all elements in a list with the following code:

```
void free_list(struct List *l)
{
    while (l != NULL) {
        free(l);
        l = l->next; // l->next is a read of deallocated memory
    }
}
```

A pointer to a deallocated piece of memory is called a *dangling pointer*.

Importance

When your program deallocates memory, heap management functions such as `malloc` and `free` can modify it. This means that if your program reads from the deallocated memory, you might get unexpected results. If a program writes to the deallocated memory, it might corrupt internal data structures used by heap management functions.

In a multi-threaded process, a memory region deallocated by thread A might be allocated by thread B. This means that if thread A writes to the deallocated memory, it might corrupt thread B's data, and if thread A reads from the deallocated memory, it might return pieces of thread B's data.

Avoidance

Having dangling pointers present in your program can be disastrous. We suggest that you structure your code so that the pointer falls out of scope immediately after the program deallocates the memory to which it points. Alternatively, set the pointer to NULL after the program deallocates the memory to which it points. Here is a much safer implementation for freeing all elements in a list:

```
void free_list(struct List *l)
{
    while (l != NULL) {
        struct List *tmp = l;
        l = l->next;
        free(tmp);
        // tmp goes out of scope here, so no need to clear
    }
}
```

Here are two other examples that demonstrate how to avoid dangling pointers:

```
void *x = malloc(X_SIZE);
...
free(x); x = NULL; // dangling pointer now cleared

{
    void *y = malloc(Y_SIZE);
    ...
    free(y);
} // dangling pointer now inaccessible
```

7: Memory area deallocated twice

DoubleCheck issues this error when your program deallocates the same memory region twice without allocating it again.

Importance

Deallocating the same memory region twice might cause heap management functions such as `malloc` and `free` to function improperly.

Avoidance

Avoid dangling pointers using techniques discussed in the Avoidance section of “6: Access into deallocated memory” on page 381.

9: Attempt to deallocate stack memory

DoubleCheck issues this error when your program deallocates stack memory.

Stack memory is memory that your program uses to hold local variables and function return addresses. It also holds objects allocated with `alloca`. The compiler generates code that manages stack memory. *Heap memory* is memory that heap management functions such as `malloc`, `free`, `new`, and `delete` use when allocating and deallocating regions. You should never pass a pointer to stack memory to a heap management function.

Importance

Deallocating stack memory causes heap management functions like `malloc`, `free`, `new`, and `delete` to function improperly.

Avoidance

Avoid using the same pointer variable to refer to both stack memory and heap memory. For example, if a variable contains pointers to heap memory 99 percent of the time and pointers to stack memory 1 percent of the time, you might make the mistake of unconditionally deallocating the memory to which the pointer points.

10: Inactive stack address potentially returned

DoubleCheck issues this error when a function returns a pointer to a local variable. For example:

```
char *int_to_string(int val)
{
    char buffer[11];
    for (int i = 9; i >= 0; i--) {
        buffer[i] = '0' + val % 10;
        val /= 10;
    }
    buffer[10] = '\0';
    return buffer;
}                                     // inactive stack memory returned!
```

Importance

When a function returns, any pointers to non-static local variables in that function become pointers to inactive stack memory. If the function returns one of these pointers and the caller dereferences it, you might get unexpected results because another function call or interrupt handler might have changed the inactive stack memory. If your program reads this memory, it might not return the desired value. If your program writes to this memory, it might corrupt stack memory that contains values critical to correct program execution.

Avoidance

Do not write functions that return pointers to local variables. One way to avoid accessing inactive stack memory is to declare the variable as static, however, this technique encourages accidental overlapping uses of the static variable. For example, if you modify the `int_to_string()` example so that `buffer` is declared `static`, this function does not correctly print two different integers:

```
void print_two_ints(int a, int b)
{
    printf("%s %s\n", int_to_string(a), int_to_string(b));
}
```

To avoid returning pointers to local variables, require callers to pass in pointers so that the caller can decide how to allocate the memory.

11: Write to potentially read-only memory

DoubleCheck issues this error when the program writes to read-only memory. For example:

```
char *get_string()
{
    return "Hello";
}
char *get_lower_case_string()
{
    char *st = get_string();
    for (char *c = st; *c != '\0'; c++)
        *c = tolower(*c);    // writing to read-only memory!
    return st;
}
```

Importance

Writing to read-only memory can cause exceptions if the memory is write-protected. Writing to read-only memory can cause unexpected future results. For example, calling `get_lower_case_string()` changes what future calls to `get_string()` will return.

Avoidance

Use the `const` keyword when declaring pointers to read-only memory. Using this keyword eliminates the chance of writing through the pointers. To correct the preceding example, define `get_string` with return type `const char *`.

12 / 13: Memory deleted using “delete[]” instead of “delete” / Memory deleted using “delete” instead of “delete[]”

DoubleCheck issues this error when your program allocates memory with `new []` and deallocates it with `delete`, or when it allocates memory with `new` and deallocates it with `delete []`.

Importance

If you mix an allocator with the wrong deallocator, your program may not call the appropriate destructor. This mismatch may not seem like a bug if, for example, a destructor is not present. It is a better coding practice to use the proper `new` and `delete` calls in case you add a destructor in the future.

Avoidance

Always use `new` and `delete` calls that match. To strictly enforce this rule on a certain class, overload the unwanted `new` or `delete` calls so that they output an error message if you use them by mistake.

14: Read of potentially uninitialized variable

DoubleCheck issues this error when a local variable is never written with a value, but is read on a particular execution path.

Importance

Relying on an uninitialized value can produce unexpected results. The uninitialized value might not cause problems 99 percent of the time, which can make reproducing failures very difficult.

Avoidance

It is good practice to always initialize local variables. This practice should not degrade performance since the Green Hills compiler is very good at eliminating unused initializers.

Warnings

Warnings indicate code that will not explicitly cause unexpected behavior, but is worth investigating. The code might not be optimal or might contain checks for problems that can arise in a different form.

8: Useless NULL-check after pointer has been dereferenced

DoubleCheck issues this warning when your program performs a NULL-check on a pointer after it has already dereferenced that pointer.

Importance

When you perform a NULL-check, it indicates that you think the pointer will be NULL in some cases. If you perform this check after dereferencing the pointer, this in turn indicates that your program will dereference a NULL pointer in those cases. Your program might exhibit unexpected behavior if it dereferences a NULL pointer.

Avoidance

To avoid this warning and related potential problems, move the NULL-check before the dereference, and use the NULL-check to avoid executing the dereference when the pointer is NULL.

15: Value set but never read / Value overwritten before being read

DoubleCheck issues this warning when your program writes a variable with a value, but no execution paths leading from this write perform a read of that variable. Execution paths leading from this write might write the variable again, destroying the first value and making it impossible to read.

DoubleCheck does not issue this warning when you clear a dangling pointer using the following code:

```
free(x) ;  
x = NULL;
```

Importance

This warning indicates that you might have intended to use the value written to the variable, so some functionality might be missing from your program.

Avoidance

Avoid writing to a variable unless you intend to use the value stored in the variable for some purpose.

16: Pointer dereferenced unconditionally after NULL-check

DoubleCheck issues this warning when your program performs a NULL-check on a pointer, and then dereferences that pointer independent of the result. To avoid producing these warnings on infeasible execution paths, assign properties to any functions that affect execution paths. For example, the following piece of code will produce a warning without the `#pragma` directive:

```
#pragma double_check no_return 0 my_error_handler
    if (x == NULL)
        my_error_handler();
    *x = 0;
```

Importance

When you perform a NULL-check, it indicates that you think the pointer will be NULL in some cases. If you dereference a pointer after a NULL-check, it indicates that you will dereference a NULL pointer in those cases. Your program might exhibit unexpected behavior if it dereferences a NULL pointer.

Avoidance

When a pointer is found to be NULL, it is good practice to prevent your program from dereferencing that pointer. You can do this by stopping execution with a call that does not return (such as `exit`, `abort`, or a user-defined error handling function), or by jumping to a point later in the control flow that bypasses the dereference.

Using Advanced Tools

Part II

The asppc Assembler

Chapter 9

This Chapter Contains:

- Running the Assembler from the Builder or Driver
- Running the Assembler Directly
- Assembler Options
- Assembler Syntax
- Expressions
- Labels
- -list File Output
- Reading the Compiler's Assembly Output

The assembler translates ASCII files containing assembly language instructions into binary files containing relocatable object code.

Running the Assembler from the Builder or Driver

We recommend that you invoke the assembler via the Builder or driver. Simply add assembly files to the Project Manager's source pane, or to the driver's command line. If your assembly file ends with **.ppc** rather than **.s**, the Builder or driver first invokes the preprocessor to take advantage of preprocessor facilities (such as **#include** and **#define**) that are not normally available to an assembly language programmer. To run the preprocessor on all assembly files (including those with a **.s** extension):



Set the **Assembler→Preprocess Assembly Files** option to **On** (**-preprocess_assembly_files**).

The Builder and driver accept a limited number of assembler-specific options directly (see “Assembler Options” on page 241). To pass the full range of assembler options listed in this chapter:



Enter the required assembler option (see “Assembler Options” on page 394) in the **Assembler→Additional Assembler Options** option (**-asm=assembler_option**).

When using the driver, you can either precede each assembler option with **-asm=**, or list multiple options separated by whitespace and enclosed within quotes. Hence the two following command lines are equivalent:

```
ccppc -asm=-stack_debug -asm=-w stack.s -c -o stack.o
ccppc -asm="-stack_debug -w" stack.s -c -o stack.o
```

Alternatively, if you have a large number of options to pass to the assembler, you can place them in a text file and enter its filename in the **Assembler→Assembler Command File** option (**-asmcmd=file**).

Generating Assembly Language Files

At times, you might want to generate and review assembly language output from a high-level language file. To do this:



From the driver, pass the **-S** option. For example, the following command line would compile **main.c** and place the assembly language output in the file **main.s**:

```
ccppc -S main.c
```

Running the Assembler Directly

The syntax for running the assembler directly is:

```
asppc [options] [input_files]
```

where *options* are assembler options. When running the assembler directly, do not use the **-asm=** option.

Assembler Options

The assembler combines each specified input file and produces a single output object module. For information about using the following options with the Builder or driver, see “Running the Assembler from the Builder or Driver” on page 392.

PowerPC-Specific Assembler Options

-b
-b1 Produces big endian output. This is the default.
-b0 Produces little endian output.
-cpu=cpu Specifies a particular PowerPC target processor. For a complete list of supported processors and the options to be passed to specify them, see “PowerPC Processor Variants” on page 114.
-noSPE Produces fatal errors for any SPE instructions. This option is only applicable to processors that support SPE, such as PowerPC 8540.
-regs Enforces that registers are not specified by numbers. Normally, the assembler accepts <code>add 3, 3, 4</code> to mean the same thing as <code>add r3, r3, r4</code> . When the -regs option is specified, the former syntax is rejected.
-SPE Allows SPE instructions on processors that support them, such as PowerPC 8540. This is the default for all processors that support SPE.

General Assembler Options

-help Prints a help message.
-I dir (an uppercase letter “i”) Searches directory <i>dir</i> for files specified in <code>.include</code> directives.

-list *[[,pagelength][,pagewidth]][=[file]]*

Generates a source listing of, by default, 64 lines per page and unlimited page width. An alternative page length and/or width can be specified.

If `=` is specified without *file*, then the listing is displayed on the standard output. If `=file` is specified, then the listing is written to *file*. If neither `=` nor *file* is specified, then the listing file is written to a new file with the same name as the output file, but with an **.lst** extension.

Examples:

- **-list**: Saves listing to **.lst** file with default page length and width.
- **-list,80**: Saves to **.lst** file with page length of 80 and default width.
- **-list=trax**: Saves to file **trax** with default page length and width.
- **-list,,110**: Saves to **.lst** file with default page length and width of 110.
- **-list,,110=trax**: Saves to file **trax** with default page length and width of 110.
- **-list=**: Displays listing on standard output with default page length and width.

-nogen

Disables source listing of macro expansions.

-o file

Sets the name of the output object file to *file*.

Without the **-o** option, the assembler produces an object file that has the name of the last specified assembly language file, with a **.o** extension. For example, **foo.o** is produced for **foo.s**.

-r

Prints a listing of symbol names in alphabetical order.

-ref

Prints a full cross reference of the symbols in alphabetical order, including the symbol name, type, file, and line defined and the file and line of each usage.

-V

Prints the assembler version number to the standard output.

-w

Disables assembler warnings.

Assembler Syntax

Identifiers

Identifiers, or symbols, are composed of letters, digits, and the following special characters: dollar sign (\$), period (.), and underscore (_). The first character of an identifier must be alphabetic, or one of these three special characters. Uppercase and lowercase letters are distinct; the identifier abc is not the same as the identifier ABC. Characters in reserved symbols, such as directives, machine instructions, and registers, are case-sensitive.

Identifiers can be up to 4096 characters in length, with all characters significant.

Examples

The following table shows some valid and invalid identifiers:

Identifier	Validity
*star	Invalid (may not contain *)
123test	Invalid (may not start with digit)
f-ptr	Invalid (may not use hyphen)
_hello	Valid
LABEL	Valid
test4	Valid

Reserved Symbols

Operator names are reserved. For a list of operators, see “Integral Expression Operators” on page 401.

The following additional identifiers are built into the PowerPC assembly language.

Identifier	Meaning
r0 — r31	integer registers
f0 — f31	floating point registers
v0 — v31	vector registers (Altivec only)

Identifier	Meaning
sr0 — sr15	segment registers (not available on the PPC500)
sp	stack pointer (r1)
cr0-cr7	condition register fields
lt, gt, eq, so, un	condition register bits

The names of the special purpose registers (SPR's) such as XER, LR, and CTR, are also reserved. Consult the appropriate microprocessor user's manual for the list of implemented SPR's.

Constants

Assembler constants can be numeric, character, or string constants.

Numeric Constants

A sequence of digits defines a numeric constant. Constants can be specified in hexadecimal, octal, or binary formats, or as floating-point numbers, by preceding the number with one of the following special prefixes:

Type	Prefix	Example
hexadecimal	0x	0xb0b
octal	0	0747
binary	0b	0b110011
floating-point	0f	0f6.02e+23

To specify a negative floating-point number, the minus sign must be placed in front of the number, but after the 0f prefix. For example, to specify the negative of the floating-point number listed in the table, use 0f-6.02e+23, not -0f6.02e+23.

String Constants

Some directives take a string constant as one or more of their arguments. A string constant consists of a sequence of characters enclosed in double quotation marks ("). A string constant can contain any ASCII character (including ASCII

null), except newline. The null character is not appended to strings by the assembler, as it is in C or C++.

Character Constants

A character constant can be used in any location where an integer constant is needed.

A character constant consists of the following items, enclosed within single quotation marks (' '):

- either a single ASCII character, or
- a backslash character (\) and one of the escape character values

A character constant is considered equivalent to the ASCII value of the character or escape sequence.

For example, the character constant ' a ' is equivalent to the decimal integer 97, and the character constant ' \r ' is equivalent to the decimal integer 13.

Character Escape Sequences

Character and string constants consist of ASCII characters. The ASCII backslash (\) is used within character and string constants to escape the quotation marks and to specify certain control characters symbolically. A backslash followed by any non-escape character is equivalent to that character (for example, \a is equivalent to a because \a is not an escape sequence).

Escape Sequence	Character	ASCII value
\0	null	0 (0x0)
\b	backspace	8 (0x8)
\t	horizontal tab	9 (0x9)
\n	newline	10 (0xA)
\v	vertical tab	11 (0xB)
\f	form feed	12 (0xC)
\r	return	13 (0xD)
\nnn	octal value <i>nnn</i>	n/a

Escape Sequence	Character	ASCII value
<code>\xnn</code>	hexadecimal value <i>nn</i>	n/a
<code>\'</code>	single quotation mark	39 (0x27)
<code>\"</code>	double quotation mark	34 (0x22)
<code>\\</code>	backslash in a constant or string	92 (0x5C)

Source Statements

An assembler source statement consists of a series of fields delimited by spaces and/or horizontal tabs, in the following format:

```
[label:] [operator [arguments]] [comments]
```

Label Field

See “Labels” on page 407.

Operator Field

The operator field starts with the first non-whitespace character after the optional label field and is terminated by the first whitespace character or line terminator encountered after the operator. An operator is any symbolic opcode, directive, or macro call. In order to avoid ambiguity with the label field, operators should not begin in column 1.

Argument Field

The argument field starts with the first non-whitespace character following the operator field and ends with a line terminator or the beginning of a comment field. Arguments qualify the opcode, directive, or macro call.

Comment Field

The comment field is optional and begins with “#”. The assembler ignores all characters to the right of the comment symbol until the end of the line.

Continuation Lines

The assembler does not support continuation lines.

Whitespace

Whitespace consists of spaces and horizontal tabs.

Line Terminators

Assembly input lines are terminated by a line feed or a carriage return and line feed combination. A carriage return or form feed is insufficient.

Expressions

Assignment Statements

An expression is assigned to a symbol by an assignment statement in one of the following forms, where *ident* is the symbol name:

- *ident* = *const-expr*

For example:

```
table = 8           # set table to 8
table = 5           # reset table to 5
```

- *ident* .equ *const-expr*

You cannot use this directive multiple times for the same symbol. For example:

```
table .equ 8        # set table to 8
table .equ 5        # try to reset table to 5 (illegal)
```

- .set *ident*, *const-expr*

For example:

```
.set table,8        # set table to 8
.set table,5        # reset table to 5
```

const-expr must be an absolute or relocatable expression (for more information, see “Expression Types” on page 406).

Integral Expression Operators

A number of operators are available to form expressions. The type `unary` indicates that the function is recognized when the operator has only a right operand. The type `binary` indicates that the operator has two operands, one to the left of the operator and one to the right.

Operator	Type	Meaning
~	unary	Bitwise-not operator.
-	unary	Negative.

Operator	Type	Meaning
+	binary	Add.
-	binary	Subtract.
*	binary	Multiply.
/	binary	Divide.
%	binary	Modulo.
&	binary	Bitwise-and operator.
	binary	Bitwise-or operator.
^	binary	Bitwise-exclusive-or operator.
=,==	binary	Equality (0 or 1).
!=	binary	Inequality (0 or 1).
>,>=,<,<=	binary	Signed compare (0 or 1): greater than, greater than or equal to, less than, less than or equal to.
:UGT:, :UGE:, :ULT:, :ULE:	binary	Unsigned compare (0 or 1): greater than, greater than or equal to, less than, less than or equal to.
<<,>>	binary	Signed shift left, signed shift right.
:USHR:	binary	Unsigned shift right (shift 0 into high bit).
:ROTR:, :ROTL:	binary	Rotate right, rotate left.

Operator Precedence

The following table lists the operators in decreasing order of precedence. The binary operators associate from left to right:

~ and - (unary)
*, /, %, <<, >>, :USHR:, :ROTR:, and :ROTL:
+ and - (binary)
=, .equ, ==, !=, <, >, <=, >=, :ULT:, :UGT:, :ULE:, and :UGE:
&
and ^

Expressions are grouped with matching parentheses ().

Relocation Expression Operators

The type `unary` indicates that the function is recognized when the operator has only a right operand. These should only be used as instruction arguments.

Operator	Type	Meaning
<code>%lo(value)</code>	<code>unary</code>	Least significant 16 bits of symbol <i>value</i> .
<code>%hi(value)</code>	<code>unary</code>	Most significant 16 bits of symbol <i>value</i> .
<code>%hiadj(value)</code>	<code>unary</code>	Most significant 16 bits of symbol <i>value</i> plus most significant bit of <code>%lo(value)</code> .
<code>%pidlo(value)</code>	<code>unary</code>	Least significant 16 bits of PID symbol <i>value</i> .
<code>%pidhi(value)</code>	<code>unary</code>	Most significant 16 bits of PID symbol <i>value</i> .
<code>%pidhiadj(value)</code>	<code>unary</code>	Most significant 16 bits of PID symbol <i>value</i> plus most significant bit of <code>%pidlo(value)</code> .
<code>%sdaoff(value)</code>	<code>unary</code>	16 bits offset of SDA symbol <i>value</i> . This relocation also modifies the base register of the encapsulated instruction to match the base register for the SDA section holding <i>value</i> . For example, the following code: <pre>lwz r3, %sdaoff (x) (r13)</pre> may have <code>r13</code> replaced with <code>r0</code> should <i>x</i> end up in ZDA.

C Type Information Operators

If you enable the `-asm3g` option (see “**Support for C Type Information in Assembly**” on page 243), you can import type information from C header files using the `.inspect` directive and use it in the following operators:

Expression	Meaning
<code>offsetof(type, ident)</code>	The offset of <i>ident</i> from the base type of <i>type</i> in bytes, where <i>type</i> is a structure tag or a typedef identifier.
<code>sizeof(type)</code>	The size of <i>type</i> in bytes, where <i>type</i> is a built-in type, a structure tag, or a typedef identifier.

When specifying a structure tag for a `sizeof()` or `offsetof()` directive, do not precede it with `struct` as you do when writing C code. For example, use `sizeof(foo)` instead of `sizeof(struct foo)`.

To specify a member M of a struct type S, use `sizeof (S->M)`. You must use `->` instead of `.`, because the assembler considers `.` a valid identifier character.

The second argument to `offsetof` must be a simple identifier. For example, the following code is not allowed:

```
offsetof(myStruct, myArrayMember[5])
```

Example 1. Using C Type Information Operators

This example shows you how to use type information in assembly directives. The following header file **foo.h** defines the type **myStruct**:

```
/* foo.h */

struct myStruct {
    char myChar;
    int myInt;
    struct {
        short myShort;
        int myArray[5];
    } nested;
};
```

The following assembly file **foo_asm.s** uses the type information for **myStruct** to define objects:

```
# foo_asm.s

# Get type information for myStruct from foo.h
.inspect "foo.h"
.data

# Use type information from myStruct to write data to objects
myStruct_size:
    .long sizeof(myStruct)
nested_size:
    .long sizeof(myStruct->nested)
myArray_size:
    .long sizeof(myStruct->nested->myArray)
myInt_offset:
    .long offsetof(myStruct, myInt)
myShort_offset:
    .long offsetof(myStruct->nested, myShort)
# The following two lines are syntax errors
#     .long sizeof(myStruct->myArray[0])
#     .long offsetof(myStruct, nested->myShort)
```

Expression Types

The primary expression types are:

Expression type	Meaning
absolute	A value of an identifier or expression that is computed by the assembler during assembly.
quoted string	A C-style character string delimited by double quotation marks is used in conjunction with a number of assembler directives. All string escape sequences defined in the C language, such as <code>\n</code> for newline, are allowed. These sequences are described in “Character Escape Sequences” on page 398.
relocatable	A relocatable expression or identifier assigns a value relative to the beginning of a particular section. These values are determined at link time. All label identifiers are relocatable values.
undefined	If an identifier is unassigned, its value cannot be determined until link time. This is an undefined external.

Type Combinations

Constants can be combined with all operators. You can combine a constant with a relocatable or absolute type using any of the operators in the following table:

Operator	Type	Meaning
+	binary	addition (if one operand to the plus operator is a constant, the result is the type of the other operand)
-	binary	subtraction (if the second operand is constant, the result is the type of the first operand. If both operands are in the same section, then the result is a constant that is the difference of the addresses)

Examples

```
4           # constant
4*(5+6)     # constant
label      # relocatable
```

Labels

A statement can begin with one or more labels. Each label is either a *named label* or a *temporary label*.

Named Labels

Named labels are identifiers followed by one or two colon characters. Labels defined with one colon are not visible outside the source module. A second colon specifies that the label is made visible external to its source file, instead of local to that file.

Temporary Labels

Temporary labels consist of a non-zero decimal number followed by a colon. Any number of these labels can be present, even if the value of the constant is repeated. A reference to a temporary label consists of the label's constant value expressed as a decimal number followed immediately, with no space, by one of the following characters: `f` or `b`:

- `f` — This reference refers to the nearest statement with the same numeric label occurring after the reference, not including the current source line.
- `b` — This reference refers to the nearest statement with the same numeric label occurring on or before the current source line.

Matching labels in the opposite direction of the one specified are not referenced.

Example 2. How to Use Temporary Labels

```
                                # timer loop
.macro loop reg
    cmplwi reg,0                # exit if reg < 0
    blt 1f                      # jmp to end of macro
2:
    addi reg,reg,-1             # decrement counter
    cmplwi reg,0
    bne 2b                      #loop until we reach zero
1:
    .endm
.global func
func:
    li r6,10
    loop r6                    # loop 10 times
    li r7,15
    loop r7                    # loop 15 times
    li r8,-5
    loop r8                    # doesn't loop
```

Current Location

The symbol `$` refers to the current location in the current section, which can be used as an alternative to creating a label when the current location must be referenced. For example:

```
sparse_table:
    .long 1,2

    # move to address that is 16 bytes from sparse_table
    .skip 16-($-sparse_table)
    .long 3,4

    # move to address that is 32 bytes from sparse_table
    .skip 32-($-sparse_table)
    .long 5,6
```

-list File Output

When you specify the **-list** option, the assembler creates an output listing file. If the **-ref** option is also given, the file includes a cross reference, as the example below illustrates. Given the following input file:

```
.title  "file.s"
.macro  li      reg, val
        lis     reg, %hi(val)
        ori     reg, reg, %lo(val)
.endm
.globl  func
func:
        li      r4, 0x12345678
        li      r5, 0x87654321
        add     r3, r4, r5
        blr
.end
```

when compiled with `asppc -list -ref file.s`, the resulting **file.lst** is:

```
file.s                               Mon Aug 25 20:57:29 2009           Page 1
Table of Contents                   Mon Aug 25 20:57:17 2009           file.s
Command Line:   asppc -list -ref file.s
Source File:    file.s
Directory:      /data7/mtm/tmp
Host OS:        Linux 2.4.18-3 #1 Thu Apr 18 07:37:53 EDT 2002
Version:        AS-PowerPC 5.0#06
Release:        Version 5.0
Revision Date:  Thu Aug  7 11:43:26 2009
```

```
file.s. . . . . 1 file.s
```

```
file.s                               Mon Aug 25 20:57:29 2009           Page 2
Table of Contents                   Mon Aug 25 20:57:17 2009           file.s
1  .title  "file.s"
2  .macro  li      reg, val
3          lis     reg, %hi(val)
4          ori     reg, reg, %lo(val)
5  .endm
6  .globl  func
7  func:
8          li      r4, 0x12345678
00000000 3c801234      8          lis     reg, %hi(val)
00000004 60845678      8          ori     reg, reg, %lo(val)
9          li      r5, 0x87654321
00000008 3ca08765      9          lis     reg, %hi(val)
0000000c 60a54321      9          ori     reg, reg, %lo(val)
00000010 7c642a14     10         add     r3, r4, r5
00000014 4e800020     11         blr
12  .end
```

```
SYMBOL TABLE CROSS REFERENCE      Mon Aug 25 20:57:29 2009           Page 3
                                   Mon Aug 25 20:57:17 2009           file.s
func          reloc file.s:        6          7*
li            macro file.s:        2*          8          9
```

The table of contents shows every `.title` and `.sbtbl` entry and page number. This is followed by the listing. Each line is shown with its line number from the original source file. Where macro expansion has occurred, the line number is repeated in the listing for several lines because all of those lines were produced by the original source line. The hexadecimal numbers on the left represent the offset from the start of the section and the binary object code produced from that source line.

The cross reference listing, if requested, comes at the end of the assembly listing. It describes each symbol defined in the program, its type, and the line

numbers on which it is used. The line that defines the symbol is marked with an asterisk (*). The line numbers refer to the original source file.

Reading the Compiler's Assembly Output

The assembly language file generated by the Green Hills compilers contains many useful comments, which are explained here. The format of these comments is subject to change without notice. In particular, additional comments might be generated by new releases of the compiler.

There are three sections of comments: a header comment at the beginning of the file, a function comment after each function, and a file comment at the end of the file.

Header Comment File

The first comment shows the command line to invoke the compiler driver, **ccppc**:

```
#Driver Command: ccppc file1.c -S
```

The next few lines provide the name of the source file, the compile-time directory, the date and time the compiler was invoked, and the name and version of the host operating system in use:

```
#Source File:    file1.c
#Directory:      /tmp
#Compile Date:   Thu Dec 13 13:45:02 2001
#Host OS:        SunOS 5.8 Generic_108528-06
```

Then the version of the compiler and the exact date of the compiler is shown:

```
#Version:        target version RELEASE VERSION
#Release:        Version version
#Revision Date:   Revision Date
#Release Date:    Release Date
```

Function Comment Section

The function comment section follows the function and lists parameters and local variables. It indicates where they are stored, either in a register, on the stack, or in static memory. However, if the program is compiled with optimization enabled, the generated code might drastically reduce variable lifetimes and make other alterations. In extreme cases, all uses of a variable in the register or memory location to which it is assigned can be optimized away.

<code>;f</code>	<code>f0</code>	<code>local</code>
<code>;c</code>	<code>r1</code>	<code>local</code>
<code>;.L12</code>	<code>.L15</code>	<code>static</code>
<code>;a</code>	<code>r0</code>	<code>param</code>
<code>;b</code>	<code>r1</code>	<code>param</code>

File Comment Section

This section lists imported and static variables declared at the file level and indicates where they are stored:

<code>#y</code>	<code>y</code>	<code>static</code>
<code>#z</code>	<code>z</code>	<code>import</code>

Source Line Comments

In addition to the three sections of comments described above, the Green Hills compilers can show the original source code in the assembly language output file in the form of comments. This is enabled with the `-passsource` option. Source lines for `#include` files are never shown. Some blank or inconsequential lines are deleted by the compiler. Otherwise, the source lines are shown exactly as they occur in the original source before preprocessor expansion.

Assembler Directives

Chapter 10

This Chapter Contains:

- Alignment Directives
- Conditional Assembly Directives
- Data Initialization Directives
- File Inclusion Directives
- Macro Definition Directives
- Repeat Block Directives
- Section Control Directives
- Source Listing Directives
- Symbol Definition Directives
- Symbolic Debugging Directives
- Miscellaneous Directives

Assembler directives are instructions to the assembler to perform various functions. The following tables provide detailed descriptions of the assembler directives available to you when using the **asppc** assembler.

Alignment Directives

```
.align bound-expr
```

Advances the location counter to the specified addressing boundary, where *const-expr* is an exponent of 2. The following table summarizes the possible addressing boundaries specified by *const-expr*.

- 0: 1 byte
- 1: 2 bytes
- 2: 4 bytes
- 3: 8 bytes
- 4: 16 bytes

The location counter advances to the next address that is a multiple of the specified addressing boundary. For example, 8, 16, 24, 32, and 40 all have an 8-byte addressing boundary. As the location counter advances, skipped addresses are filled with zeros for the `.data` section and `nop` instructions (`0x60000000`) for the `.text` section.

If you place `.align` immediately following a section directive (for example, `.data` or `.text`), the linker ensures that the entire section begins on the specified addressing boundary. For example, suppose an assembly source file contains the following:

```
.data
.align 3
```

When the source file is linked, the starting address of the `.data` section is aligned on an eight byte boundary. If the addressing boundary is defined in more than one source file being linked, the linker aligns the final section to the largest boundary found in the individual sections. For example, suppose two source files, **file_1.s** and **file_2.s**, contain the following:

- **file_1.s:**

```
.data
.align 3
```

- **file_2.s:**

```
.data
.align 4
```

When **file_1.s** and **file_2.s** are linked, the `.data` section is aligned on a 16 byte boundary.

Conditional Assembly Directives

The `.if const-expr` directive starts a block of assembly statements that are assembled only if *const-expr* is true. The `.endif` directive terminates the conditional block of assembly statements.

Within an `.if` / `.endif` block, the `.else` and `.elseif` directives provide alternative assembly statements that are assembled only if the `.if const-expr` directive evaluates to false.

```
.if x==2
.ascii "if directive is TRUE"
.elseif x-6
.ascii "elseif directive is TRUE"
.else
.ascii "both if and elseif are FALSE"
.endif
```

You can nest conditional blocks within conditional blocks.

By default, when the assembler creates a source listing, it always lists `.if` / `.endif` assembly statements, but lists the corresponding object code only if the block is assembled. That is, the assembler produces and lists object code only if *const-expr* evaluates to true.

<code>.else</code> Assembles the subsequent block of assembly statements if the previous <code>.if</code> directives evaluate to false.
<code>.elseif <i>const-expr</i></code> Assembles the subsequent block of assembly statements if the previous <code>.if</code> directives evaluate to false and the <code>.elseif</code> 's <i>const-expr</i> evaluates to true. The assembler must be able to resolve <i>const-expr</i> to a constant without referring to information within the conditional block of code that follows the <code>.elseif</code> directive.
<code>.endif</code> Terminates the code block that began with the previous <code>.if</code> directive.
<code>.if <i>const-expr</i></code> Starts a conditional block of assembly statements; the assembler generates object code for these statements only if <i>const-expr</i> evaluates to true. The assembler must be able to resolve <i>const-expr</i> to a constant without referring to information within the <code>.if</code> / <code>.endif</code> conditional block of code.

Data Initialization Directives

The following directives evaluate expressions and place successive values of the specified type in the assembly output.

Use `.offset` to advance the location counter to the address where you want to place data.

<code>.ascii "string"</code>	Evaluates a C-style string enclosed in double quotation marks (" ") and places the characters in successive bytes. The delimiting double quotation mark characters and the implicit terminating null are discarded.
<code>.asciz "string"</code>	Evaluates a C-style string enclosed in double quotation marks (" ") and places the characters in successive bytes. Unlike the <code>.ascii</code> directive, the terminating null is placed with the string. The delimiting double quotation mark characters are discarded. Identical to <code>.strz</code> .
<code>.byte const-expr, const-expr, ...</code>	Stores the values of constant expressions in successive bytes. Each constant expression must be in the signed range -128 to 127 or in the unsigned range 0 to 255.
<code>.double flt-expr, flt-expr, ...</code>	Stores the values of floating-point expressions as successive 64-bit IEEE-754 floating-point values. Each floating-point expression must be within double-precision range.
<code>.float flt-expr, flt-expr, ...</code>	Computes the values supplied by the floating-point expressions and produces successive 32-bit IEEE-754 floating-point values. Each floating-point expression must be within single-precision range.
<code>.long const-expr, const-expr, ...</code>	Stores the values of constant expressions as successive 32-bit data. The expressions can be absolute expressions, relocatable expressions, or undefined external identifiers. The actual values of relocatable and undefined external identifiers are supplied at link time. The value of each expression must be in the signed range -2147483648 to 2147483647 or the unsigned range 0 to 4294967295.
<code>.offset const-expr</code>	Places subsequent data at the address offset <code>const-expr</code> from the beginning of the current section in the current compilation unit. Attempting to place subsequent data prior to the current offset will result in an error.

```
.short const-expr, const-expr, ...
```

Stores the values of manifest expressions as successive 16-bit data. The manifest expressions must be in the signed range -32768 to 32767 or the unsigned range 0 to 65535.

```
.skip const-expr
```

Generates *const-expr* bytes of zero data.
Identical to `.space`.

```
.space const-expr
```

Generates *const-expr* bytes of zero data.
Identical to `.skip`.

```
.strz "string"
```

Evaluates a C-style string enclosed in double quotation marks (" ") and places the characters in successive bytes. Unlike the `.ascii` directive, the terminating null is placed with the string. The delimiting double quotation mark characters are discarded.

Identical to `.asciz`.

File Inclusion Directives

```
.include "file"
```

Inserts the contents of *file* at the location of this directive. The assembler searches for *file* in the current working directory, then searches in directories specified by the **-linc_directory** compiler driver option.

Macro Definition Directives

A `.macro`/`.endm` block defines the contents of a macro. When the name of the macro appears in the same assembly file, the assembler replaces the name of the macro with the macro's contents. This replacement is called *macro expansion*.

A macro might reference another macro. In this case, the assembler processes the enclosed macro when it expands the enclosing macro. Macros cannot call themselves recursively.

The `.macro` directive marks the beginning of a macro and is followed by the name of the macro and a list of parameters. Next, the body of the macro contains assembly statements that are included in an assembly file during macro expansion. To end the body of the macro, place a `.endm` directive as the first symbol on a line.

If you want the assembler to prematurely terminate macro expansion when a certain condition is true, enclose the `.exitm` directive within an `.if / .endif` block in the body of the macro. The `.exitm` directive prematurely terminates macro expansion.

To create a label that changes each time a macro is called, define `label\@` in the definition of the macro, where `label` is a valid identifier. During macro expansion, the assembler expands `label\@` into `label#`, where `#` is a number unique to a single macro invocation.

The following example defines two macros, `trythis` and `log_and`:

```
.macro trythis parm1
lwz r1,parm1*2 (r4)
.endm
.macro log_and parm1 parm2
lwz parm1, parm2 (r4)
.endm
```

To call a macro, put the name of the macro and its arguments in an assembly file. The macro must have been previously defined in the assembly file. You must specify a value for every parameter in the macro.

During macro expansion, the macro's parameters are replaced by the values specified after the macro name. The concatenation operator `><` can be used to concatenate two parameters or a parameter with a symbol. The resulting assembly statement is not scanned for further parameter matches.

For example, to define a macro that concatenates two parameters into a label, enter the following:

```
.macro make_word_point_to_itself val1, val2
val1 >< val2:
.long val1 >< val2
.endm
```

To call the macros `trythis` and `log_and` (defined previously) and place the resulting assembly statements in the `.text` section, and call `make_word_point_to_itself` and place the data in the `.data` section, enter the following:

```
.text
trythis 16
log_and r1, 0
log_and r1, 2

.data
.space 20
make_word_point_to_itself label, 15
```

The assembler performs macro expansion and generates the following assembly:

```
.text
lwz r1,32(r4)
lwz r1,0(r4)
lwz r1,2(r4)

.data
.space 20
label15:
.long label15
```

<code>.endm</code>
Terminates the body of a macro whose definition began with a preceding <code>.macro</code> directive. The <code>.endm</code> directive must be the first symbol on its line and cannot have a label.
<code>.exitm</code>
Causes the assembler to exit a macro without further expanding its body of assembly statements. Use within a <code>.if / .endif</code> block to specify a condition under which you do not want a macro expanded beyond that point.
<code>.macro name [param1, param2, ...]</code>
Begins the definition of a macro, where <i>name</i> is a string representing the name of the macro and <i>param1</i> is the first parameter that the macro accepts. Parameters must be separated by a comma or whitespace.

Repeat Block Directives

The following directives specify a block of assembly statements that repeat *const-expr* times.

```
.rept const-expr  
...  
.endr
```

Repeat blocks can occur within repeat blocks. In this case, the enclosed repeat block is expanded once for each expansion of the enclosing block. The repeat count of an inner block is evaluated each time the block is expanded.

You can define a repeat block within a macro definition as long as the repeat block is completely enclosed within the macro.

<pre>.endr</pre>
Terminates a repeat block that began with a preceding <code>.rept <i>const-expr</i></code> directive. The <code>.endr</code> directive must be the first symbol on its line and cannot have a label.
<pre>.rept <i>const-expr</i></pre>
Begins a block of assembly statements that repeats <i>const-expr</i> times, where <i>const-expr</i> resolves to a constant number.

Section Control Directives

These directives direct assembly output into the specified section.

<pre>.bss</pre>
Changes the active section to the <code>.bss</code> section, which contains uninitialized data. Data that follows this directive is placed in the <code>.bss</code> section.
<pre>.data</pre>
Changes the active section to the <code>.data</code> section. Data that follows this directive is placed in the <code>.data</code> section.
<pre>.note "<i>string</i>" [, "a" "b" "v" "w" "x"]</pre>
Changes the active section to an ELF note section. The new section is called <i>string</i> . The attribute letters <code>a</code> , <code>b</code> , <code>v</code> , <code>w</code> , and <code>x</code> function as they do with the <code>.section</code> directive.
<pre>.org <i>const-expr</i></pre>
Creates an absolutely located data section at address <i>const-expr</i> . The name of the section is the section's address, with a prefix of <code>.org</code> . The MULTI Debugger cannot debug code placed at an address with a <code>.org</code> directive. Therefore, we recommend that you use the <code>.section</code> directive to declare code and data in a section instead, and then define the address of the section in a linker directives file.
<pre>.previous</pre>
Changes the active section back to the one in use before the most recent section control directive.

`.rodata`

Changes the active section to the `.rodata` section, which is used for read-only data. Data that follows this directive is placed in the `.rodata` section.

`.sdata`

Changes the active section to the `.sdata` section, which is used for SDA. Data that follows this directive is placed in the `.sdata` section.

`.sdata2`

Changes the active section to the `.sdata2` section, which is used for Read-Only SDA. Data that follows this directive is placed in the `.sdata2` section.

```
.section "string" [, "a" | "b" | "v" | "w" | "x" ]
```

Directs assembly output into the section named *string*, which is defined in terms of the optional attributes as listed below.

- **a**: the section should have memory allocated for it; that is, it should not be used solely for debugging or for symbolic information.
- **b**: the section will have BSS semantics. Although normal data directives such as `.long` and `.byte` are allowed in a `.bss` section, all of the values specified in those directives are discarded by the assembler. In the ELF output file, the assembler records only the size of the section, and its contents are omitted. When the section is downloaded to the target, space is allocated for the section, but no data is downloaded to this section. Instead, the startup code is responsible for initializing all bytes in the section to zero.
- **v**: the section contains VLE code. Specifying this flag does not enable the VLE instruction set. The `.vle` directive must be used for this purpose. `v` sets the `SHF_PPC_VLE` flag bit in the ELF section header flags (`sh_flags`) field for this particular section.
- **w** — the section is writable.
- **x** — the section contains executable code.

If none of these letters are specified, then no attributes are set. This is appropriate only for sections containing debugging or other information not intended to be part of the final linked file. Sections that are intended to be part of the final linked output should have at least the `a` attribute. After the attributes are set they cannot be specified again.

The standard sections have predefined attributes as follows:

- `.text` (Program code): "ax"
- `.data` (Initialized data): "aw"
- `.rodata` (Read-only initialized data): "a"
- `.bss` (Uninitialized data): "awb"

This example creates a section called `.mytext` with allocation and execute attributes:

```
.section ".mytext", "ax"
```

```
.text
```

Changes the active section to the `.text` section. Assembly code and data that follows this directive is placed in the `.text` section.

Source Listing Directives

The following directives control the format and content of the source listing produced by the assembler. They do not affect how the assembler generates object code.

Use the **-list** compiler driver option to produce a source listing. For example, to have the assembler produce a source listing **file.lst** when it assembles **file.s**, enter:

```
ccppc file.s -list
```

<code>.eject</code> Puts a form feed (^L) into the source listing.
<code>.gen</code> Causes macros following this directive to be included in the source listing. Identical to the <code>.list .macro</code> directive.
<code>.list [.if] [.include] [.list] [.macro] [.macrodef] [.rept]</code> Controls what the assembler includes in the source listing. All these elements are included by default, and these directives are only used to reverse a previously given <code>.nolist</code> directive of the same type. • <code>.list</code>: Enables source listing. • <code>.list .if</code>: Lists all branches of conditional blocks (<code>.if/.else/.endif</code>). • <code>.list .include</code>: Displays include files. • <code>.list .list</code>: Controls printing the <code>.list</code> directive itself. • <code>.list .macro</code>: Expands all macros. • <code>.list .macrodef</code>: Displays all macro definitions. • <code>.list .rept</code>: Controls printing the <code>.rept</code> directive itself. This directive fully expands all repeat blocks.
<code>.nogen</code> Prevents macros in the sections following this directive from being included in the source listing. Identical to the <code>.nolist .macro</code> directive.

`.nolist [.if][.include][.list][.macro][.macrodef][.rept]`
Causes the assembler to exclude information from the source listing until a corresponding `.list` directive is encountered.

- `.nolist`: Disables source listing.
- `.nolist .if`: Lists only those conditional branches (`.if ...`) for which code is actually generated.
- `.nolist .include`: Prevents the display of include files in the sections following this directive.
- `.nolist .list`: Prevents the display of `.list` and `.nolist` directives.
- `.nolist .macro`: Prevents the display of macro expansions in the sections following this directive.
- `.nolist .macrodef`: Prevents macro definitions from displaying in the source listing.
- `.nolist .rept`: Prevents the expansion of repeat blocks in the sections following this directive.

`.sbttl "string"`

`.subtitle "string"`

Inserts *string* as the subtitle at the top of each page in the source listing. The double quotation marks are discarded.

`.title "string"`

Inserts *string* as the title at the top of each page in the source listing. The double quotation marks are discarded.

Symbol Definition Directives

The following directives define the type and value of an identifier.

<code>.comm <i>ident</i>, <i>const-expr</i> [, <i>const-expr</i>]</code> Assigns the specified identifier <i>ident</i> to a common area of <i>const-expr</i> bytes in length, and causes <i>ident</i> to be visible externally. If <i>ident</i> is not defined by another relocatable object file, the linker assigns space for the identifier in the <code>.bss</code> section. The optional <i>const-expr</i> specifies the variable alignment in bytes.
<code>.dsect</code> Begins a block for defining constants. Within a <code>.dsect</code> / <code>.end</code> block, only labels and <code>.skip</code> directives are permitted. The symbolic names used for labels are defined to have values equal to the number of bytes from the beginning of the <code>.dsect</code> block. No space is actually allocated; only the symbolic names in the label fields are defined.
<code>.end</code> Terminates a block of constants that began with a preceding <code>.dsect</code> directive.
<code>.extern <i>ident</i></code> Identical to <code>.globl</code> . It is common for <code>.globl</code> to cause symbols defined in the current module to be externally visible, while <code>.extern</code> explicitly declares symbols defined in other modules. The assembler does not require this usage, however.
<code>.global <i>ident</i></code> <code>.globl <i>ident</i></code> Causes the identifier <i>ident</i> to be visible externally. If the identifier is defined in the current module, this directive allows the linker to resolve references by other modules. If the identifier is not defined in the current module, the assembler resolves it externally.
<code>.inspect "file.h"</code> Parses the C header file <i>file.h</i> to get type information. You can use this information in the <code>offsetof()</code> and <code>sizeof()</code> operators (see “C Type Information Operators” on page 403). This directive is available only when the -asm3g option is enabled (see “ Support for C Type Information in Assembly ” on page 243).
<code><i>ident</i> .equ <i>const-expr</i></code> Defines the constant value <i>const-expr</i> to the symbol <i>ident</i> . After a symbol is defined with a <code>.equ</code> directive, it cannot be redefined.
<code>.lcomm <i>ident</i>, <i>const-expr</i> [, <i>const-expr</i>]</code> Assigns the specified identifier <i>ident</i> to an area in <code>.bss</code> of <i>const-expr</i> bytes in length. The optional <i>const-expr</i> specifies the variable alignment in bytes. The <code>.lcomm</code> directive is similar to the <code>.comm</code> directive, except that <i>ident</i> is not exported.


```
.lsbss ident, const-expr [, const-expr]
```

Assigns the specified identifier *ident* to an area in `.sbss` of *const-expr* bytes in length. The optional *const-expr* specifies the variable alignment in bytes. The `.lsbss` directive is similar to `.sbss`, except that *ident* is not exported.

```
.sbss ident, const-expr [, const-expr]
```

Assigns the specified identifier *ident* to a common area of *const-expr* bytes in length and causes *ident* to be visible externally. If *ident* is not defined by another relocatable object file, the linker assigns space for the identifier in the `.sbss` section. The optional *const-expr* specifies the variable alignment in bytes.

```
.set ident, const-expr
```

Defines the constant value *const-expr* to the symbol *ident*. You can use multiple `.set` directives to change the value of the same symbol.

```
.weak ident
```

Makes the symbol weak. If the *ident* cannot be located in any module at link time, the linker sets the symbol's value to zero.

Symbolic Debugging Directives

These directives are used for symbolic debugging.

```
.file "string"
```

Stores source filename *string* in the object file symbol table. The *string* filename must be delimited by double quotation marks. Use this directive no more than once per file.

```
.fsize [func,] const
```

Specifies that the frame size for the current function is *const* bytes in size. This information allows **gstack** to compute accurate frame sizes for paths that involve calls to assembly functions.

You must place this directive within the scope of the current function, before you define the next function. The optional *func* parameter can be used to specify the current function's name.

```
.scall [func,] dest
```

Specifies that the current function makes a static call to *dest*. This information helps **gstack** compute the maximum stack size that the program might use. It also helps the Debugger generate a call graph.

To indicate that the current function makes no static calls, use `__leaf__` for *dest*.

You can use multiple `.scall` directives to specify multiple calls for a single function, however, if you use `__leaf__`, you must use just one `.scall` directive for the current function.

You must place this directive within the scope of the current function, before you define the next function. The optional *func* parameter can be used to specify the current function's name.

```
.size ident, const-expr
```

Specifies the size, *const-expr*, of the function or data object *ident*. To successfully debug an assembly language source file, it should either be assembled with the **-g** driver option, (which outputs full debug symbols) or else a `.size` directive should be used to specify the size of each function.

The `.size` directive should appear at the end of the function, just after the last assembly language statement in the function. It should be of the form:

```
.size ident, $-ident
```

where *ident* is the name of the function. The expression `$-ident` will be equal to the size of the function in bytes, allowing the creation of correct debugging information.

Use this directive when defining each symbol for function deletion to work properly.

```
.type ident, [ @function | @object ]
```

Specifies whether the symbol *ident* is a function or a data object. The `.type` directive is usually used in conjunction with the `.size` directive to define a function or data object.

Use this directive when defining each symbol to get the best debugging information, and for function deletion to work properly.

Miscellaneous Directives

```
.error "string"
```

Causes the error "*string*" to be emitted to standard error output.

```
.ident "string"
```

Adds *string* to the `.comment` section within the object file. The `.comment` section is not allocatable; it does not occupy memory on the target.

<pre>.need ident</pre> Defines a relationship between the function <i>ident</i> and the current function. The linker will not delete the function <i>ident</i> as long as the current function is used, even if <i>ident</i> is not used. Suppose a source file contains the following: <pre>myfunc: .need otherfunc # Insert assembly function here</pre> The linker will not delete the function <i>otherfunc</i>, even if it is unused, because it is needed by the function <i>myfunc</i>.
<pre>.novle</pre> Switches from PowerPC VLE code to standard PowerPC code.
<pre>.nowarning</pre> Disables warnings.
<pre>.vle</pre> Switches from standard PowerPC code to PowerPC VLE code. See the following example of mixing VLE and standard PowerPC code:
<pre>.warning ["string"]</pre> Enables warnings. If “<i>string</i>” is specified, prints the warning “<i>string</i>” to standard error output.

Example 1. VLE and Standard PowerPC code

```

        .section ".vletext", "vax"
        .align 1
        .vle
_start::
        se_bl    foo
        se_b     _start

        .text
        .align 2
        .novle
foo:
        li      r3, 4
        blr

```


The elxr Linker

Chapter 11

This Chapter Contains:

- Running the Linker from the Project Manager or Driver
- Linker-specific Options
- Specifying the Program Entry Point
- Configuring the Linker with Linker Directives Files
- Symbol Definitions
- Advanced Linker Features

The Green Hills **elxr** linker combines ELF object files and libraries into a single executable program suitable for downloading onto your target. To perform this task, the linker:

1. combines all input files into a single output program that contains all necessary objects.
2. assigns a memory address to each of the objects.
3. resolves relocations by replacing references to each object with the memory address it assigned to that object.

When the linker combines object files, it uses a *section map* to determine which sections in each object file to place into each program section. It also reads the section map to assign program sections to memory blocks or addresses. The linker reads a *memory map* to determine the size and location of memory blocks on your target. The linker reads section maps, memory maps, and options from linker directives (**.ld**) files (see “Configuring the Linker with Linker Directives Files” on page 437).

Running the Linker from the Project Manager or Driver

When you add source and object files to the **Project Tree** in the Project Manager, or set linker options in the **Build Options** window, the Project Manager changes the command line options that it passes to the driver when you build your project. If you do not use the Project Manager, but you use the driver to compile and link your program in multiple steps, you must pass the exact same set of options for each step.



The driver recognizes many different options and passes them to the linker. For a list of these driver options, see “Linker Options” on page 244. Additionally, the linker accepts a small set of options that the driver does not recognize. For a list of these options, see “Linker-specific Options” on page 434. Be careful when using linker-specific options, because they are not processed by the driver and may conflict with driver options.

To pass one of these linker-specific options when invoking the driver, use the **Linker→Additional Linker Options (before start file)** option (see “**Additional Linker Options (before start file)**” on page 248).

When using the command line to invoke the driver, you can either precede each linker-specific option with **-lnk=**, or list multiple options separated by a whitespace and enclosed within quotes. For example, the following command lines are equivalent:

```
ccppc -lnk=-keep=foo -lnk=-keep=bar hello.c  
ccppc -lnk="-keep=foo -keep=bar" hello.c
```

Linker-specific Options

The following table lists linker options that are not recognized by the driver. For information about options that the driver passes to the linker, see “Linker Options” on page 244.

-allabsolute
Imports all absolute symbols, not just those required. For use with the -A driver option (see “ Import Symbols ” on page 254).
-allow_bad_relocations
Emits only warnings when encountering bad or unknown relocations. The default behavior is to generate errors.
-append
Suppresses warnings concerning object file sections that do not appear in the section map.
-earlyabsolute
Imports absolute symbols before searching libraries. For use with the -A driver option (see “ Import Symbols ” on page 254).
-extractall
Forces all object files from libraries to be included in the executable. This option is different from the -extractall= driver option in that it affects all libraries, not just the one specified (see “ Force All Symbols From Library ” on page 254).
-extractweak
Pulls in object files from libraries if they provide a definition for an undefined weak reference. Normally, object files are only pulled in from a library if they provide a definition for an undefined non-weak reference.
-help
Displays a list of the most commonly used linker options.
-Help
Displays a list of all available linker options.
-keep=function
Prevents the deletion of <i>function</i> when you specify the -delete option. If you are using wholeprogram optimizations, you may also need to use “ Symbols Referenced Externally ” on page 181.
-many_segments
Prevents multiple sections from being allocated to a single segment. By default, multiple sections can be allocated to a segment if they are contiguous.

-no_append

Promotes warnings concerning sections that do not appear in the section map to errors.

-no_crom

Disables the use of compressed ROM. Sections that are specified as compressed ROM copies (through use of the attribute `CROM`) are treated as if they were specified as ROM copies using the `ROM` attribute. For more information about the `CROM` attribute, see “Using Section Attributes” on page 451.

-nosegments_always_executable

By default, the linker sets the execute bit (`p_flags & 0x1`) on all ELF program headers, meaning that code can be executed from any segment. If you pass this option to the linker, it sets the program header execute bit only for segments that have at least one input section that has its execute bit (`sh_flags & 0x4`) set.

-no_uncompressed_copy

By default, the linker creates a notes section called `.UNCOMPRESSEDsecname` for each compressed ROM section `secname`. For example, the linker would create `.UNCOMPRESSED.text` for the `.text` section. This notes section contains an uncompressed copy of the section's contents for use by the debugger. This notes section does not increase the size of the program image in memory, but it does increase the size of the ELF file itself. Use **-no_uncompressed_copy** to disable the generation of `.UNCOMPRESSED` notes sections, resulting in smaller ELF files and limiting your ability to debug CROM sections.

-prog2

Produces exactly two segments: one to contain sections `.text` and `.rodata`, and the other to contain `.data` and `.bss`.

-T file.ld

Reads additional options, memory maps, and section maps from the linker directives file `file.ld`. You do not need to pass this option manually, because the driver determines the correct **-T** options to pass to the linker.

-unalign_debug_sections

Overrides the section alignment specified in object files and forces `.debug` sections to have a 1-byte alignment.

-underscore

Adds an underscore to the beginning of all symbols created at link time.

-v

Prints verbose information about the activities of the linker, including the libraries it searches to resolve undefined symbols.

-V

Displays the version number and copyright information for **elxr**.

-w

Suppresses linker warnings.

-wrap sym

Resolves external references to *sym* to `__wrap_sym`, and external references to `__real_sym` to *sym*. This can be used to provide your own wrapper function that can be called instead of the original function from a library.

For example, if a third party library defines:

```
void critical_section(void);
```

you can check the calls to `critical_section()` by passing **-wrap critical_section** to the linker, and then define the following:

```
void __wrap_critical_section(void)
{
    assert(is_safe_to_call_critical_section());
    __real_critical_section();
}
```

Specifying the Program Entry Point

Use the **Linker→Start Address Symbol** option (**-e**) to specify the program entry point.

If you do not set this option, the linker sets the entry point to an item in the following list (in descending order of preference):

- The value of the symbol `_start`, if present
- The value of the symbol `start`, if present
- The value of the symbol `_main`, if present
- The value of the symbol `main`, if present
- The zero address

Configuring the Linker with Linker Directives Files

This section explains how to use linker directives (**.ld**) files to define program sections and assign those sections to memory blocks. The names, address, and sizes of memory blocks are defined in a *memory map*. The names, attributes, and locations of sections are defined in a *section map*. This section also describes linker directives file syntax, describes the four linker directives, and discusses some advanced topics:

- “Linker Directives File Syntax” on page 438
- “Setting Linker Options with the OPTION Directive” on page 438
- “Setting Defaults with the DEFAULTS Directive” on page 439
- “Defining a Memory Map with the MEMORY Directive” on page 440
- “Defining a Section Map with the SECTIONS Directive” on page 442
- “Modifying your Section Map for Speed or Size” on page 458
- “Customizing the Run-Time Environment Program Sections” on page 459

When you create a project with the **Project Wizard**, it generates default linker directives files for various program layouts for your board. If you want to write your own linker directives file, we recommend that you copy one of these default files and modify it. For information about these default files, see “Working with Linker Directives Files” on page 63.

Use caution when removing program sections from the default section map, because the Green Hills run-time environment might use them (see “Customizing the Run-Time Environment Program Sections” on page 459).



Note If you specify multiple linker directives files, **elxr** processes those files in the order you specify them. If you define a memory map and section map in separate files, you must specify the file containing the memory map first.

Linker Directives File Syntax

This section covers general linker directives file syntax and expressions. For information about syntax for a specific directive, see the documentation for that directive.

- Expressions — include all standard C assignment operators with their normal precedence, as well as operators that have side effects. The following sections indicate where you can use expressions in each directive.
- Comments — begin with the number symbol (#) or two forward slashes (//). If you use the **Linker→Preprocess Linker Directives** option, you must use forward slashes when writing comments.
- The dot symbol (.) — evaluates to the linker’s current position in laying out memory blocks or sections. You can change the value of the dot symbol using expressions.

Setting Linker Options with the **OPTION** Directive

You can include the linker options listed in “Linker-specific Options” on page 434 in a linker directives file using the following syntax:

```
OPTION ("option...")
```

You cannot use expressions in the **OPTION** directive.



Note This directive is provided for backwards compatibility; we recommend that you specify linker options using the driver.

Example 1. Writing an **OPTION** Directive

This example defines an **OPTION** directive that specifies the **-extractweak** and **-many_segments** options:

```
OPTION ("-extractweak -many_segments")
```



Note You can also use the `OPTION` directive to specify which files you want to link, using the following syntax:

```
OPTION ("file")
```

Setting Defaults with the `DEFAULTS` Directive

Defaults are strings that you can use in place of absolute values when defining section and memory maps in a linker directives file. Specify defaults with the following syntax:

```
DEFAULTS {name=value...}
```

To set the default `foo` to the value `0x2000`, add the following line:

```
DEFAULTS {foo=0x2000}
```

You can specify multiple defaults by putting each default on its own line:

```
DEFAULTS {  
    foo = 0x2000  
    bar = 0x4000  
}
```

You can use expressions in the `DEFAULTS` directive that include defaults set earlier in the directive:

```
DEFAULTS {  
    foo = 0x2000  
    // Set bar to 0x4000  
    bar = foo + 0x2000  
}
```

When you specify an amount of memory, you can suffix the number of bytes with `M` to designate megabytes and `K` to designate kilobytes. You can also prefix the number with `0x` to specify the amount in hexadecimal.



Note If you specify a default with the `DEFAULTS` directive, you can change its value at link time by passing the `-C` option (see “**Define Linker Constant**” on page 253).

Example 2. Writing a DEFAULTS Directive

This example specifies a DEFAULTS directive that defines two defaults:

```
DEFAULTS {  
  // Set heap_reserve to 1048576  
    heap_reserve    = 1M  
  // Set stack_reserve to 524288  
    stack_reserve   = 512K  
}
```

Defining a Memory Map with the MEMORY Directive

A memory map names and defines memory blocks on your target. Specify each of the following elements to define a memory block:

```
MEMORY  
{  
  memory_block: ORIGIN=origin_expression, LENGTH=length_expression  
  ...  
}
```

where:

<i>memory_block</i>
Specifies the name of a memory block.
<i>origin_expression</i>
Specifies the starting address of the memory block.
<i>length_expression</i>
Specifies the length of the memory block.

Expressions in the MEMORY directive can include:

- defaults set in the DEFAULTS directive.
- the dot symbol (.). In this context, the dot symbol evaluates to the first address available after the end of the previous memory block.

If you pass a memory map to the linker, only the memory blocks named and defined in it can be referenced in an associated section map.

Example 3. Writing a MEMORY Directive

In this example, the MEMORY directive defines a memory map with two blocks:

```
DEFAULTS {
    heap_reserve    = 1M
    stack_reserve   = 512K
}

MEMORY
{
    fast_memory : ORIGIN = 0x80000000, LENGTH = 10M
    slow_memory  : ORIGIN = 0xbfc00000, LENGTH = 10M
}
```

This memory map defines the memory blocks `fast_memory`, which begins at `0x80000000`, and `slow_memory`, which begins at `0xbfc00000`. They are both 10 megabytes in size.

Reserved Memory Blocks

Sometimes the default Green Hills linker directives files contain reserved memory blocks. Typically, this is done to avoid using memory that preconfigured software on your target, such as a monitor, already uses.

Reserved memory blocks are just like any other memory blocks, except that you should not allocate sections to them.

```
MEMORY {

    // 10MB of dram starting at 0x80000000

    dram_rsvd1 : ORIGIN = 0x80000000, LENGTH = 32K
    dram_memory : ORIGIN = .,          LENGTH = 10M - 32K
    dram_rsvd2 : ORIGIN = .,          LENGTH = 0

    // 10MB of flash starting at 0xbfc00000

    flash_rsvd1 : ORIGIN = 0xbfc00000, LENGTH = 32K
    flash_memory : ORIGIN = .,          LENGTH = 10M - 32K
    flash_rsvd2 : ORIGIN = .,          LENGTH = 0

}
```

Defining a Section Map with the SECTIONS Directive

A section map maps each section in each object file to a program section. The section map also maps program sections to memory blocks. Format a section map as follows:

```
SECTIONS
{
  secname [start_expression] [attributes] : [{ contents }] [> memory_block,... | >.]
  ...
}
```

where:

<i>secname</i>	Specifies name of the section.
<i>start_expression</i>	Specifies the starting address of the section. If you specify both <i>start_expression</i> and <i>memory_block</i> , <i>start_expression</i> takes precedence.
<i>attributes</i>	Specifies section attributes. You can use expressions in attribute parameters. For a list of section attributes, see “Using Section Attributes” on page 451.
{ <i>contents</i> }	Specifies the contents of the section. It contains <i>section inclusion commands</i> that dictate which sections from which object files to place into the section, and assignments that create or modify objects. You can use an unlimited number of commands and assignments.
> <i>memory_block</i> ,... >.	Allocates the section to the memory block named <i>memory_block</i> , defined in the memory map. If you specify both <i>start_expression</i> and <i>memory_block</i> , <i>start_expression</i> takes precedence.

Expressions in the SECTIONS directive can include:

- defaults defined in the DEFAULTS directive.
- memory blocks defined in the MEMORY directive.
- the dot symbol (.). In this context, the dot symbol evaluates to the linker’s current position in laying out objects and sections.
- ELF symbols defined in the object files passed to the linker.
- the functions listed in “SECTIONS Directive Functions” on page 449.

Specifying the Location of a Section

The following list describes where the linker allocates a section in memory depending on how you specify *start_expression* and *memory_block*:

- If you specify *start_expression*, the linker allocates the section starting at the address *start_expression*, whether or not you specify *memory_block*.
- If you specify just *memory_block*, the linker allocates the section starting at the first available address in *memory_block*. If the section does not fit, the linker returns an error.
- If you do not specify *start_expression*, but you specify the dot symbol for *memory_block*, the linker allocates the section directly after the previous section, in the same memory block. If the section does not fit, the linker returns an error.
- If you do not specify *start_expression*, but you specify a comma-separated list of memory blocks for *memory_block*, the linker allocates the section to the first specified memory block with enough remaining space to hold it.
- If you do not specify either *start_expression* or *memory_block*, the linker allocates the section directly after the previous section, in the same memory block. If the section does not fit, the linker allocates the section to the first memory block with enough remaining space to hold it.

No matter whether the linker determines the start address from *start_expression* or *memory_block*, it might increase the address to satisfy the alignment requirements of its contents. For example, say you have the following directive:

```
SECTIONS {  
    .text    0x10001 :  
}
```

The final executable's `.text` section might be located at `0x10004` instead of `0x10001` if the `.text` section of the first object file the linker places in this section requires 4-byte alignment.

Example 4. Writing a SECTIONS Directive

In this example, the `SECTIONS` directive defines a section map that tells the linker where to place the `.text`, `.data`, `.bss`, and custom `.mydata` sections in memory on your target:

```
DEFAULTS {
    heap_reserve   = 1M
    stack_reserve  = 512K
}

MEMORY
{
    fast_memory : ORIGIN = 0x80000000, LENGTH = 10M
    other_memory : ORIGIN = 0xbfc00000, LENGTH = 8M
    slow_memory  : ORIGIN = 0xf2c00000, LENGTH = 10M
}

SECTIONS
{
    // Place .text into fast_memory. Fail if it does not fit.
    .text : > fast_memory
    // Place .data into fast_memory after .text, or into other_memory if .data
    // cannot fit into fast_memory, or into slow_memory if .data cannot fit
    // into other_memory. Otherwise, fail.
    .data :
    // Place .bss after .data. Fail if it does not fit in the same memory block.
    .bss : >.
    // Place .mysection starting at the first available address in fast_memory.
    // If it does not fit, place it into other_memory instead.
    .mysection : fast_memory, other_memory
}
```

Specifying Multiple Memory Blocks for a Section

If you specify a single memory block for a section, and the section does not fit in that memory block, the linker returns an error. If you specify multiple memory blocks, the linker places the section in the first specified memory block with enough remaining space to hold it (see Example 4 on page 443).

On targets that support Small Data Area (SDA) optimization, be careful when using a comma-separated list of memory blocks with SDA sections. For example, it is dangerous to write:

```
SECTIONS {  
    ...  
    .sbss      : > fast_but_small_memory, slow_but_large_memory  
    .sdata     : > fast_but_small_memory, slow_but_large_memory  
    ...  
}
```

because `.sdata` and `.sbss` must be contiguous in memory. When using this section map, the linker might put `.sbss` in the `fast_but_small_memory` block and `.sdata` in the `slow_but_large_memory` block (or vice versa, if `.sbss` does not fit in the `fast_but_small_memory` block but `.sdata` does). Therefore, you should not use this syntax on SDA sections. See “Special Data Area Optimizations” on page 137 for more information.

Writing Section Inclusion Commands

Section inclusion commands are part of the `{ contents }` parameter in a line in the section map. By default, the linker places each section from each object file into the program section with the same name. If there is no program section with the same name, the linker creates a new program section at the end of the current section map and emits a warning (to silence or promote these warnings, see the **-append** and **-no_append** linker options).

If you want to change this behavior, use section inclusion commands to dictate which sections from which object files the linker should place into each section of the program. The linker executes section inclusion commands in the order that it encounters them in the linker directives file. They use the syntax *filename(section_name)*:

```
// Place the .data section from foo.o into .mydata  
.mydata      :{ foo.o(.data) }  
// Then, place .data sections from all remaining object files into .data  
.data       :
```

Separate section inclusion commands with spaces, and write them in the order that you want the linker to execute them:

```
// Place the .data section from foo.o and bar.o into .mydata  
// The foo.o .data section goes first  
.mydata      :{ foo.o(.data) bar.o(.data) }
```

If you want to place sections from a specific object file in a specific library into a program section, include the library name using the form *library_name (filename (section_name))* :

```
// Place the .data section from foo.o contained within lib.a into .mydata
.mydata      :{ lib.a(foo.o(.data)) }
```



Note Do not include the full path to the library; use just the library name. Library names are case-sensitive.

Using Wildcards in Section Inclusion Commands

Use an asterisk (“*”) in place of *filename* in a section inclusion command to specify all object files passed to the linker:

```
// Place .data sections from all object files into .data
.data      :{ *(.data) }
```

Because the program section and the object file sections have the same name, you do not need to specify a { *contents* } argument in this case, and the following syntax is sufficient:

```
// Place .data sections from all object files into .data
.data      :
```

If the program section and object file sections have different names, you must specify a { *contents* } argument:

```
// Place .data sections from all object files into .mydata
.mydata    :{ *(.data) }
```

You can also use wildcards inside *filename* and *section_name*. An asterisk (“*”) matches multiple characters and a question mark (“?”) matches any single character. When you use a wildcard in this fashion, surround the section inclusion command in double quotes (“”):

```
// Place .text sections from all object files matching "code_*.o" into .mytext
.mytext    :{ "code_*.o(.text)" }
```

If you have multiple sections in your object files that have a similar naming convention, you can use wildcards in a section inclusion command to assign them all to the same section in the output executable:

```
// Place all sections matching ".text_*" into .mytext
.mytext          :{ *(".text_*") }
```

You can also use wildcards when specifying a library:

```
// Place .data sections from object files contained within lib.a into .mydata
.mydata          :{ lib.a(*(.data)) }
// Place .data sections from all other object files into .data
.data            :
```

Using the COMMON Section in Section Inclusion Commands

The `.bss` section of a fully linked executable contains all the data from the `.bss` section of each included object file. It also contains uninitialized global data from the COMMON section. To instruct the linker to place COMMON data in a different section from `.bss` data, use the COMMON keyword in a section inclusion command:

```
// Place all COMMON data into .mybss
.mybss           :{ *(COMMON) }
// Place all .bss data into .bss
.bss             :
```

To allocate only the COMMON data from the **main.o** object file to `.mybss`, enter the following:

```
.mybss           :{ main.o(COMMON) }
```

You can also do this with the `.sbss` section and `SMALLCOMMON`, and the `.PPC.EMB.sbss0` section and `ZEROCOMMON`.

Using Assignments

The syntax for an assignment is *symbol* = *expression*; . You can set a symbol to an absolute address or an address relative to a section.

- To set a symbol to an absolute address, write the assignment on its own line in the SECTIONS directive. The linker evaluates *expression* to an address, and sets the symbol to that address. The dot symbol (.) evaluates to the absolute address at the end of the last defined section.
- To set a symbol to an address relative to a section, write the assignment inside the { *contents* } parameter for that section. The linker evaluates *expression* to an offset *n*, and sets the symbol to an address *n* bytes from the

beginning of the section. The dot symbol evaluates to the current offset from the beginning of the section.

You can assign addresses to symbols whether or not they exist:

- If you assign an address to a non-existent symbol, the linker creates that symbol.
- If you assign an address to an existing symbol, the linker reassigns that symbol without moving any data associated with that symbol. For example, if you assign an address to `main`, the linker causes relocations to `main` to point to the new address, but it does not move the actual code for `main` to that address.

To add padding at the current position in a section, increment dot. If you decrease the value of dot, the linker issues an error. The following example sets the symbol `foo` to the beginning of `.mysection` and adds 4 bytes of padding between `foo` and `bar`:

```
.mysection : { foo = .; . += 4; bar = .; }
```

Without this padding, the linker would assign the same address to both `foo` and `bar`.

Example 5. Working with Absolute and Relative Assignments

This example shows you how to set symbols to absolute addresses and addresses relative to a section.

```
SECTIONS {
    // Set alias to the same absolute address as printf
    alias = printf;

    // Assignments end with semicolons, but section inclusion commands do not.
    .text : { alpha = 0; *(<.text>) beta = .; *(<.text2>) bad = printf; } > dram_memory

    // Set gamma to address 0
    gamma = 0;

    // Set delta to the end of the .text section
    delta = .;
}
```

`alias`, `gamma`, and `delta` are assigned absolute addresses, because they are not defined inside the contents of any section. `alpha`, `beta`, and `bad` are assigned addresses relative to the beginning of `.text`, because they are defined inside its `{ contents }` parameter.

The linker assigns addresses to the symbols in `.text` as follows:

- `alpha` is located at offset 0 from the start of `.text`.
- `beta` is located in `.text` after the `.text` input sections, at the same address as the first object in the `.text2` input sections.
- `bad` is located at an offset that is `printf` bytes from the start of `.text`. It is not located at the same address as `printf`, because the assignment is relative to the section. For example, if `.text` starts at `0x4000` and `printf` is located at `0x4020`, then `bad` is located at `.text+0x4020`, or `0x8020`.

SECTIONS Directive Functions

The linker recognizes the following functions during expression evaluation in the `SECTIONS` directive. Their names are case-insensitive.

<code>absolute(expr)</code>
Given a section-relative offset value, <code>absolute</code> returns the absolute address by adding the address of the containing section to <code>expr</code> . It is an error to use <code>absolute</code> outside of the braces that specify a section's contents.
<code>addr(sectname)</code>
Returns the memory address of the section or memory block <code>sectname</code> .
<code>align(value)</code>
Returns the current position (.) aligned to a <code>value</code> -byte boundary. <code>value</code> must be a power of two. This is equivalent to: <code>(. + value - 1) & ~(value - 1)</code>
<code>alignof(sectname)</code>
Returns the alignment of the section <code>sectname</code> .
<code>endaddr(sectname)</code>
Returns the memory address of the current end of the section <code>sectname</code> , or the end of the memory block <code>sectname</code> .
<code>error("string")</code>
Generates a linker error, displaying <code>string</code> , the current section's name and address, and the current section offset.
<code>final(finalexpression [,earlyexpression=0])</code>
Evaluates the given expression only during the final layout pass. The linker evaluates the optional second argument during all preceding passes.

<code>isdefined(symbol)</code>
Returns 1 if <i>symbol</i> exists and is defined, 0 otherwise.
<code>isweak(symbol)</code>
Returns 1 if <i>symbol</i> is a weak global symbol, 0 otherwise.
<code>memaddr(sectname)</code>
See <code>addr</code> .
<code>memendaddr(sectname)</code>
See <code>endaddr</code> .
<code>min(value1, value2)</code> <code>max(value1, value2)</code>
Returns the minimum or maximum, respectively, of the two values supplied.
<code>pack_or_align(value)</code>
Returns the current position (.) aligned such that the section cannot span a page boundary of size <i>value</i> . Generally, you only use this expression as the <i>start_expression</i> for a section map. It is equivalent to: <code>((. % value) + sizeof(this_section) > value ? align(value) : .)</code>
<code>provide(name = expr)</code>
If the symbol <i>name</i> is not yet defined, then this is equivalent to <i>name</i> = <i>expr</i> . Otherwise, it is equivalent to <i>expr</i> .
<code>sizeof(sectname)</code>
Returns the current size of the section <i>sectname</i> , or the size of the memory block <i>sectname</i> .

Example 6. Incrementing the Address of an Existing Symbol

Depending on your target and the optimizations you enable, the linker might perform program layout multiple times. When necessary, use the `final()` function to ensure that the linker evaluates an expression during the final layout only.

For example, say that if the symbol `func` exists, you want to increment its address by one. The following assignment might cause the linker to increment the address of `func` multiple times:

```
// Incorrect - might increment multiple times
.text : { isdefined(func) ? (func += 1) : 0; }
```

Instead, use the `final()` function to ensure that the linker increments the address of `func` once:

```
// Correct - only increments during the final layout
.text : { final(isdefined(func) ? (func += 1):0); }
```

Using Section Attributes

Use the following attributes to configure section behavior:

ABS
Sets a flag in the output file that indicates that this section has an absolute address and should not be moved. Program loaders and other utilities that manipulate the output image should not include this section in any movement related to position independent code or position independent data.
ALIGN(<i>n</i>)
Aligns the start address of the section to the byte boundary given by <i>n</i> .
AT(<i>expr</i>) LOAD(<i>expr</i>)
These attributes specify an expression <i>expr</i> that indicates the address where the section will be loaded in memory. This is only relevant if the address the section is linked to is different than the address where the section should be loaded. Note that AT, unlike LOAD, must be placed after the <code>:</code> in the section map. This is mainly used for enhanced compatibility with GNU linker directives files.

CLEAR

NOCLEAR

Sets or removes the CLEAR attribute of the section. If CLEAR is set, the linker makes an entry for the section in the run-time clear table. Startup code often uses this table to initialize memory blocks to zero (see “Run-Time Clear and Copy Tables” on page 472).

CLEAR is specified implicitly on .bss, .PPC.EMB.sbss0, and .sbss sections; all other sections default to NOCLEAR. If you direct the linker to include COMMON, ZEROCOMMON, or SMALLCOMMON data in any other section, you should specify the CLEAR attribute for that section. For suggested uses, see Example 8 on page 455.

FILL(*value*)

Fills the section with the byte pattern *value*. The section must have the CLEAR attribute set either explicitly, or by default. If the CLEAR attribute is not set, the linker ignores the FILL attribute.

MAX_ENDADDRESS(*address*)

Causes the linker to generate an error if the section extends beyond *address* during final layout.

MAX_SIZE(*size*)

Causes the linker to generate an error if the section exceeds *size* bytes in length.

MIN_ENDADDRESS(*address*)

Instructs the linker to pad with zeros, if necessary, to ensure that the section extends to at least *address*.

MIN_SIZE(*size*)

Instructs the linker to pad with zeros, if necessary, to ensure that the section is at least *size* bytes in length.

NOCHECKSUM

Instructs the linker to output the section without a checksum. This overrides the effect of the **-checksum** option for an individual section.

NOLOAD

Instructs the linker to output the section as SHT_NOBITS. The linker allocates the section into the target image, but discards its contents.

PAD(*value*)

Places *value* bytes of padding at the beginning of the section. This is equivalent to specifying padding at the beginning of the section contents.

For example, the following two definitions are equivalent:

```
.stack pad(0x10000) : { }  
.stack              : { . += 0x10000; }
```

```
ROM(sect_name)
```

```
CROM(sect_name)
```

```
ROM_NOCOPY(sect_name)
```

These attributes allocate the contents of a section named *sect_name* to ROM at link time, while reserving space for the data to be copied to RAM. The `ROM` and `CROM` options instruct the linker to put an entry in the real-time ROM copy table and compressed ROM copy table, respectively, so that the Green Hills startup code copies the section from ROM to RAM.

`CROM` compresses its copy by as much as 40% to 50%, and thus can greatly reduce the amount of ROM required for your application. For more information, see “Copying a Section from ROM to RAM at Startup” on page 456.

`ROM_NOCOPY` behaves like the `ROM` section attribute except that it does not instruct the linker to put an entry in the real-time ROM copy table. As a result, the Green Hills startup code will not copy the section from ROM to RAM. Use this option when you want to use your own startup code for copying a section from ROM to RAM.

```
SHFLAGS(expr)
```

Sets the `sh_flags` field of the section’s ELF header to the given expression *expr*.

For example, to specify that a section should be given the `SHF_ALLOC` and `SHF_WRITE` flags, use:

```
SHFLAGS(0x3)
```

For more information, see “Section Attribute Flags” on page 811.

Example 7. Setting the Size of the Stack and Heap

This example demonstrates how to set the size and alignment of the `.stack` and `.heap` sections using section attributes:

```
DEFAULTS {
    heap_reserve    = 1M
    stack_reserve   = 512K
}

MEMORY
{
    fast_memory : ORIGIN = 0x80000000, LENGTH = 10M
    slow_memory : ORIGIN = 0xbfc00000, LENGTH = 10M
}

SECTIONS
{
    .text                : > fast_memory
    .data                : > .
    .bss                 : > .
    .heap                ALIGN(8) PAD(heap_reserve) : > .
    .stack               ALIGN(8) PAD(stack_reserve) : > .
}
```

The `.heap` section has two attributes, `ALIGN` and `PAD`. `ALIGN(8)` directs the linker to place `.heap` at an address that is a multiple of 8, regardless of the size of the sections before it. `PAD(heap_reserve)` uses the `heap_reserve` default, and effectively sets the size of the heap to 1 MB.

The `.stack` section has the same two attributes as the heap.

All of the sections following `.text` use the dot (`.`) in place of a memory block, so the linker places them consecutively in `fast_memory` after `.text`. If all of the sections do not fit into `fast_memory`, the linker returns an error.

Example 8. Using CLEAR and NOCLEAR

This example shows you how to use the CLEAR and NOCLEAR attributes to control whether or not the linker puts an entry for a section into the run-time clear table. For more information about the table, and how startup code uses it to initialize sections to zero, see “Run-Time Clear and Copy Tables” on page 472.

The following line specifies the .bss section:

```
.bss :
```

By default, the linker makes an entry for .bss in the run-time clear table. When the program starts, it will initialize all data in this section to zero.

This line describes a section named .mybss:

```
.mybss CLEAR : { * (COMMON) }
```

{ * (COMMON) } directs the linker to put COMMON data in this section (see “Using the COMMON Section in Section Inclusion Commands” on page 447). CLEAR directs the linker to make an entry in the clear table for .mybss. When the program starts, it will initialize data in this section to zero.

You can also use CLEAR for the heap:

```
.heap CLEAR PAD(0x100000) :
```

By default, the linker does not make an entry in the run-time clear table for sections other than .bss, .PPC.EMB.sbss0, and .sbss. CLEAR overrides this behavior, so the linker makes an entry for .heap. PAD(0x100000) adds 1 MB of padding to the beginning of .heap, effectively setting the size of the heap to 1 MB. When the program starts, it will allocate 1 MB of RAM for this section, and initialize it to zero.

Copying a Section from ROM to RAM at Startup

Use the ROM and CROM attributes to control whether or not the linker puts an entry for a section into one of the run-time copy tables. Startup code uses these tables to determine which sections to copy to RAM at startup. For more information about these tables, see “Run-Time Clear and Copy Tables” on page 472.

When defining a new section with the ROM or CROM attributes:

- The name of the new section must be `.ROM.sect`, where *sect* is the name of the existing section. *sect* must not include any periods (`.`).
- The ROM or CROM attribute must also specify *sect*.

Do not specify section contents or any other attributes for the new section; the new section inherits the attributes and contents of the existing section, and the existing section becomes a placeholder in RAM. You cannot create multiple ROM or CROM copies of the same section.

For example, to place the `.data` section in ROM with instructions to have it copied to RAM at startup, include the following lines in your section map:

```
.data                                : > RAM_memory_block
...
// Create a copy of .data in ROM and
// create an entry in the run-time ROM copy table
.ROM.data      ROM(.data) : > ROM_memory_block
```

Startup code copies the ROM image of `.ROM.data` into the `.data` section, located in RAM.

To place the `.text` section in compressed ROM with instructions to have it copied to RAM at startup, include the following lines in your section map:

```
.text                                : > RAM_memory_block
...
// Create a copy of .text, compress it in ROM, and
// create an entry in the run-time compressed ROM copy table
.CROM.text    CROM(.text) : > ROM_memory_block
```

Startup code decompresses the compressed ROM image `.CROM.text` and copies the decompressed image into the `.text` section, located in RAM.

If the placement of a section that is not a ROM copy or compressed ROM copy is dependent on the size of a CROM section, you might get a link-time error. Furthermore, relocations should not reference any symbols, including `__ghsbegin_section` or `__ghsend_section`, whose locations depend on the size of a CROM section.

You can disable the effects of the CROM attribute by passing the option **-no_crom** on the command line (see “Linker-specific Options” on page 434). Sections marked as CROM are still copied to ROM but are not compressed.

Example 9. Placing CROM Sections

The following section map generates errors because the placement of `.rodata` depends on the size of `.CROM.data` and `.CROM.text`:

```
.data                : >RAM
.text                :
...
.CROM.data    CROM(.data)  : >ROM
.CROM.text    CROM(.text)  :
.rodata                :
```

To correct this example, rearrange the sections so that only `CROM.text` depends on the size of `.CROM.data`:

```
.data                : >RAM
.text                :
...
.rodata                : >ROM
.CROM.data    CROM(.data)  :
.CROM.text    CROM(.text)  :
```

Modifying your Section Map for Speed or Size

The **Project Wizard** (see “Creating A New Project” on page 22) provides a variety of default linker directives files with section maps that either maximize the speed or minimize the size of your program by varying the number of sections copied to RAM at startup. You can control this feature by manually editing your section map.

Maximizing Execution Speed

Because access times for RAM are better than for ROM on most hardware, your program will generally execute faster if all of its sections are copied from ROM to RAM at startup. The Green Hills startup code copies from ROM to RAM those sections identified with the ROM attribute (see “Copying a Section from ROM to RAM at Startup” on page 456).

If you want to use custom code instead of the Green Hills startup code to copy sections from ROM to RAM, see “Customizing the Run-Time Environment Program Sections” on page 459 and “Run-Time Clear and Copy Tables” on page 472.

Minimizing RAM Usage

To minimize RAM usage, you must ensure that as few program sections as possible are copied to RAM at startup.

If a section does not change at run time, you do not need to copy it to RAM. Although you generally must copy `.data` to RAM, the Green Hills PowerPC compiler can determine whether certain data items do not change at run time, and so can be retained in ROM. In order to take advantage of this capability, create a section called `.rodata`. The linker places user-defined `const` variables into the `.rodata` section. When you declare a variable with the keyword `const`, the value becomes a constant, ensuring that the linker places it in the `.rodata` section.

The `.rodata` section should be placed in the link map adjacent to the program `.text` section, causing this data to be put into ROM along with the program text.

If you use the SDA optimization (see “Special Data Area Optimizations” on page 137), some data that would normally go into the `.rodata` section might be placed in the `.sdata2` section.

If you use the ZDA optimization (see “Special Data Area Optimizations” on page 137), some data that would normally go into the `.rodata` section might be placed in the `.PPC.EMB.sdata0` section.

Customizing the Run-Time Environment Program Sections

The following material describes the special program sections that are created for and maintained by the Green Hills run-time environment system (the libraries **libsys.a** and **libstartup.a** and the module **crt0.o**). These sections appear in all the default linker directives files provided with your distribution, and their contents are generated automatically, so you should not explicitly add to them.



Note Any attempt to explicitly place text or data into the following sections is likely to produce unexpected results. When creating custom-named sections, you must take care not to use any of the names of these special sections. For more detailed descriptions of the functions that use these linker directives and special sections to set up the run-time environment, see “Customizing the Run-Time Environment Libraries and Object Modules” on page 751. You can also look at the comments in the customizable source files, which are located in the **src/libsys** and **src/libstartup** directories.

.fixaddr and .fixtype

The compiler creates `.fixaddr` and `.fixtype`.

These two sections contain information that enables the Green Hills startup code to relocate PIC/PID initializers of data variables. The compiler generates data in the `.fixaddr` and `.fixtype` sections, if needed, when using PIC and/or PID. The default Green Hills run-time libraries might already have information in these sections, because many of these libraries are always built with PIC and PID.

These two sections contain read-only data and can be placed in ROM. Failure to include these sections in the linker directives file might cause the linker to add them to the end of the section list, which might cause the following warning:

```
[elxr] warning: section .fixaddr from libsys.a(ind_crt1.o)
isn't included by the section map;
    appending after last section.
    add to section map or use -append to append without warning
```

If your program has dynamic memory that expands past the end of the last specified section in a section map, it might be fatal to append sections to the end of the section map, because those sections might be overwritten. Therefore, you should include these sections in section maps.

.heap

The linker creates the `.heap` section with the information from your section map. It specifies the size and location of the run-time heap. This section is required when using the Green Hills run-time libraries for dynamic memory allocation. A reference to the special linker symbol `__ghsbegin_heap`, which denotes the beginning of the `.heap` section, is located in the **`ind_heap.c`** module of the **`libsys.a`** library.

If you do not include a `.heap` section in your linker directives file, and your program pulls in `ind_heap.o`, you will get one or more linker errors, such as the following:

```
[elxr] (error) unresolved symbols: 1
'__ghsbegin_heap' from libsys.a(ind_heap.o)
```

If you are not using the Green Hills run-time libraries, you can omit the `.heap` section from your linker directives file.

The Green Hills run-time libraries ensure that dynamic memory allocation does not overrun the specified `.heap` area. The libraries do so by returning an error code from `sbrk()` that causes `malloc()` and other memory allocation routines to return a `NULL` pointer. Your code should always check for erroneous return values when calling dynamic memory allocation routines. You can customize the Green Hills `sbrk()` routine, located in **`ind_heap.c`**, to avoid using the `.heap` section.

.rodata

The compiler creates the `.rodata` section. Some Green Hills compilers place read-only data (such as data declared with the C `const` specifier and string

constants) into a read-only section called `.rodata`. By convention, you place `.rodata` with other sections that can be placed in ROM. Failure to include a `.rodata` section in the section map may cause the linker to append it to the end of the section map. If you compile your program with either **-pic** or **-pid**, it does not use this section.

ROM Sections

When the linker encounters a section directive that has the ROM attribute, it creates a read-only copy of the section. A section that contains the ROM attribute must be named `.ROM.sect`, where `.sect` is the name of the section that is copied from ROM into RAM (see “Copying a Section from ROM to RAM at Startup” on page 456). The Green Hills startup code, **ind crt0.c** (see “ind crt0.c” on page 755), copies the data from the `.ROM.sect` section to the `.sect` section when copying initialized data from ROM to RAM.

If you want to use your own custom code to copy a section from ROM to RAM at program startup, use the `ROM_NOCOPY` attribute when defining the ROM section. When you use this attribute, the linker will not create an entry in the copy table, and the Green Hills startup code will not copy the section from ROM to RAM. This attribute behaves similarly to the `ROM` and `CROM` attributes.

If the custom code uses the copy table, use the `ROM` attribute. Then, disable the Green Hills code that copies sections from ROM to RAM by modifying or deleting code in **ind crt0.c**.

If you do not need to copy sections from ROM to RAM, do not include `.ROM.sect` sections or use the `ROM`, `CROM`, or `ROM_NOCOPY` attributes in your linker directives file.

.sdabase

For completeness, all section maps should include `.sdabase`, regardless of whether or not the target supports the SDA optimization.

If SDA is implemented, the `.sdabase` section is required. In this case, place the `.sdabase` section just prior to the start of the sections that make up the SDA. The Green Hills debug servers initialize the SDA base register with the address of `.sdabase` section (a dummy, zero-sized section). The Green Hills startup code initializes the SDA base register by referencing the predefined

symbol `__ghsbegin_sdatabase` to obtain the location of the `.sdatabase` section.

.secinfo

`.secinfo` is a special section output by the Green Hills linker that contains information about the section layout of programs. The startup code, `ind crt0.c`, uses this information to determine which sections must be cleared (for example, `.bss` sections) and which sections must be copied (for example, ROM sections) at program startup. This section is read-only and thus can be placed in ROM. Failure to include this section in the link map causes the linker to append it to the end of the section map. The following three section flags control the information placed in the `.secinfo` section:

- `a` — allocated for downloading
- `w` — writable
- `x` — executable

For example:

```
.section ".foo", "aw"  
.section ".bar", "ax"
```

Consult the extensive comments in the `ind crt0.c` file for further documentation on the automatic copy/clear feature.

The linker allows a program to obtain the locations, sizes, and types of its sections at run time. If the symbol `__secinfo` is referenced but not defined in a program, the linker fills in additional section information in the `.secinfo` section and sets the `__secinfo` symbol to point to it. The `indsecinfo.h` header file in the `src/libsys` directory of your distribution has a declaration for this structure and a sample program that shows its use.

.stack

`.stack` specifies your intended stack area and size, although there is no general mechanism for ensuring that the stack will stay within the limits specified by the `.stack` section. Green Hills startup code (when running in stand-alone mode) and debug servers initialize the stack pointer of a process based on this `.stack` section. When building programs to run under an

operating system that does not allow user-specification of the stack area, an empty `.stack` section can still be included in the section map to resolve references in the startup code. The reference to the `.stack` area in the Green Hills startup code (**crt0.o**) is used only when programs are in stand-alone mode (that is, when they are not downloaded through the MULTI Debugger). Use of the `.stack` section in the startup code can also be customized by editing and rebuilding the **crt0.ppc** assembly module. The Debugger also allows the initial stack pointer to be specified differently with each download of a program via the special `_INIT_SP` variable; use of `_INIT_SP` supersedes use of a `.stack` section to locate the initial stack pointer.

.syscall

`.syscall` contains the run-time library code for Green Hills emulation of system calls. This section must be located in RAM if you use the Green Hills run-time libraries for system call emulation. When system call emulation is not being handled in this way, you can modify **ind_dots.ppc** to remove its dependency on the `.syscall` section (see “`ind_call.ppc`, `ind_dots.ppc`” on page 756).

Symbol Definitions

The linker recognizes several symbols that, if referenced but not defined in the source code, are given particular addresses corresponding to the final image.

Beginning, End, and Size of Section Symbols

When the linker performs final symbol resolution for a fully linked output file, it recognizes that certain undefined symbols refer to memory addresses in the final section map. The names of these symbols are `__ghsbeginsection`, `__ghsendsection`, and `__ghssizesection`, where *section* is the name of each section in the output file, with any dot (.) characters changed to underscores (“_”).

For example, for a section named `.text`, the symbols `__ghsbegin_text` and `__ghsend_text` resolve to the absolute addresses of the beginning and end of the section, while `__ghssize_text` refers to its size.

For a section map containing these lines:

```
SECTIONS
{
    .bss1 0x400000:
    .bss:
}
```

and this portion of **main.c** program code:

```
extern char __ghsbegin_bss1[], __ghssize_bss1[];

void foo() {
    memset(__ghsbegin_bss1, 0, (int)__ghssize_bss1);
    __ghsbegin_bss[0] = 0xff;
}
```

if the size of section `.bss1` is `0x100`, then the linker resolves the following labels:

`__ghsbegin_bss1` to `0x400000`

`__ghsend_bss1` to `0x400100`

`__ghssize_bss1` to `0x100`

`__ghsbegin_bss` to `0x400100`

End of Function Symbols

When the linker performs final symbol resolution for a fully linked output file, it recognizes that symbols named `__ghs_eofn_function`, where *function* is a function name, refer to the memory addresses of the end of functions in the final executable. For example, for function **foo**, the symbol `__ghs_eofn_foo` resolves to the virtual address of the end of that function.

For example:

```
extern void __ghs_eofn_foo(void);
int foo(int x) { return x+1; }
int main() {
    printf("Size of foo is %d\n", (int)__ghs_eofn_foo - (int)foo);
    return 0;
}
```

If the function **foo** is at address 0x100000 and has size 0x20 bytes, then the linker resolves `__ghs_eofn_foo` to be 0x100020.

Linker Generated Tables

When the linker performs final symbol resolution for a fully linked output file, it recognizes undefined symbols named `__ghstable_tablename` as tables. When it encounters one of these symbols, the linker resolves the symbol to a table that it allocates to the read-only `.rodata` section. The linker fills the table with the addresses of all defined symbols named `__ghsentry_tablename_entryname`, and then terminates it with a NULL pointer. The order of the entries within the table is unpredictable. If the name of an entry symbol matches multiple tables that the linker is filling, the entry is added to the table with the longest matching name.



Note `__ghstable_tablename` is not supported with Position Independent Code (PIC) or Position Independent Data (PID).

In the following example, `__ghstable_resetvectors` is an array of pointers to functions. The array contains the addresses of `__ghsentry_resetvectors_foo()` and `__ghsentry_resetvectors_bar()`. The function `reset_all()` has the effect of calling both functions.

```
extern void (*__ghstable_resetvectors[]) (void);

int foo = 5;
void __ghsentry_resetvectors_foo(void)
{
    foo = 0;
}

double bar = 2.0;
void __ghsentry_resetvectors_bar(void)
{
    bar = 0.0;
}

void reset_all()
{
    int i;
    for (i = 0; __ghstable_resetvectors[i] != 0; ++i) {
        (__ghstable_resetvectors[i]) ();
    }
}
```

Forcing The Linker to Pull In a Module From a Library

Normally, the linker pulls in an object file from a library only when that file provides a definition for an unresolved symbol that is not weak. You can use the **-extractall** and **-extractweak** options to change this behavior globally. However, there might be cases when you want the linker to pull in particular object files from a library when that library is added on the command line, but you do not want the linker to pull in all of the object files from that library.

The linker searches all libraries added on the command line for symbols beginning with `__ghsalwaysimport` or `__ghsautoimport`. If an object file in a library defines a symbol whose name begins with `__ghsalwaysimport`, the linker will always pull in that object file. If an object file defines a symbol whose name begins with `__ghsautoimport`, the linker will pull in that object file unless the symbol has already been defined by another object file. The distinction between `__ghsalwaysimport` and `__ghsautoimport` may be useful when dealing with common symbols (see “Allocation of Uninitialized Global Variables” in “C/C++ Data Allocation” on page 227 for more information about the **--commons** option). That is, if multiple modules define a common symbol named `__ghsalwaysimport_foo`, the

linker will pull them all in, but if multiple modules define a common symbol named `__ghsautoimport_foo`, the linker will only pull in the first such module it encounters.

Deleting Unused Functions

The **-delete** option instructs the linker to remove functions that are not referenced in the final executable. The linker iterates to find functions that do not have relocations pointing to them and eliminates them. If a symbol is referenced by a relocation in the first pass, it can still be eliminated in a subsequent pass if the referencing symbol is eliminated. Note that the **-delete** option can increase the time and memory required for linking.



Note If you are hand-coding assembly and intend to use the **-delete** option, you must properly identify assembly labels as functions or objects with the proper size.

The linker supports **-delete** only for object files produced by the Green Hills assembler. Other assemblers might produce object files that do not preserve information required for **-delete** to function properly.



Tip In cases where a symbol should never be deleted, use the **-u** driver option, **-keep=funcname** linker option, or the `.need` assembler directive.

The `.need` assembler directive introduces an explicit dependency between two symbols (usually functions) so that the linker keeps one function if another function is used. For example, use this directive when you have an assembly function that does not return, but instead falls through into the next function in the file.

The linker never deletes symbols preserved by these methods, even if the symbols appear unused. It is useful to preserve symbols such as interrupt table entry points, non-standard entry-points, or other portions of code not reached by normal control transfer within the program.

Compiler-generated profiling instrumentation or debugging information might create references to unused functions, preventing the linker from deleting them. See **-ignore_debug_references** in “**Link-Time Ignore Debug References**” on page 252 for more information.

In addition to the **-delete** option, **-uvfd** enables the deletion of C++ virtual functions. **-uvfd** requires that all modules being linked are compiled with Green Hills compilers. We recommend that you enable this option with the **-Olink** flag, rather than specify **-uvfd** explicitly (see Linker Optimizations in “Optimization Options” on page 176).

Advanced Linker Features

Code Factoring

Code factoring is a link-time optimization that reduces overall program size by removing redundant segments of code from object files. It removes these segments by using subroutine calls and tail merges. The standard libraries provided with the Green Hills tools for PowerPC are all prepared for code factoring.

On most programs, Code Factor decreases execution speed due to increased branching. The optimization should only be enabled when minimal program size is the primary goal.

To enable code factoring:



Set the **Linker**→**Linker Optimizations**→**Code Factoring** option to **On** (-codefactor).



Note Code factoring might cause DWARF debugging information to be out of sync with the program in memory.

Example 10. Basic Code Factoring

To factor code from source files **foo.c** and **bar.c**, use:

```
ccppc -codefactor -o hello foo.c bar.c
```

This command generates two object files, **foo.o** and **bar.o**. They are prepared for code factoring, and then the driver passes them to the **elxr** linker, which performs the optimization and links them into an executable, **hello**.

Example 11. Controlling the Scope of Code Factoring

The linker performs code factoring on a per-section basis. For example, if you have the following in your section map:

```
SECTIONS {  
    .footext : { foo1.o(.text) foo2.o(.text) }  
    .bartext : { bar1.o(.text) bar2.o(.text) }  
}
```

- The linker factors code from the `.text` sections of both `foo1.o` and `foo2.o`, and places the optimized code into the `.footext` section.
- The linker factors code from the `.text` sections of both `bar1.o` and `bar2.o`, and places the optimized code into the `.bartext` section.

This behavior can be useful when you want to prevent code factoring from creating dependencies between sections that must remain independent.

Example 12. Code Factoring and Incremental Linking

Code factoring runs during the final stage of linking, when the linker produces an executable file. If you incrementally link your files before producing an executable, pass the `-codefactor` option at each stage to ensure that the linker preserves the necessary relocation and symbol information. For example, to generate a relocatable object file **mod.o** from object files **foo.c** and **bar.c**, enter the following:

```
ccppc -codefactor -relobj -o mod.o foo.c bar.c
```

This command generates the relocatable object file **mod.o**, and preserves all symbol and relocation information. To subsequently generate a code factored executable, enter the following:

```
ccppc -codefactor mod.o -o hello
```

This command produces the code-factored executable, **hello**.

Example 13. Partial Code Factoring

It is possible to apply code factoring to only some of the modules that are linked into the final executable. If **foo.c**, **bar.c**, and **baz.c** are to be linked together to produce the executable **hello**, but only **bar.c** and **baz.c** are to be code factored, first generate an object file from **foo.c**, as follows:

```
ccppc -c foo.c
```

This command produces an object file named **foo.o**, which is excluded from code factoring in any subsequent link.

Next, generate object files from **bar.c** and **baz.c**, using the `-codefactor` option, as follows:

```
ccppc -codefactor -c bar.c baz.c
```

This command produces the object files **bar.o** and **baz.o**, which are prepared for subsequent code factoring.

Finally, generate an executable, as follows:

```
ccppc -codefactor foo.o bar.o baz.o -o hello
```

This command produces an executable, **hello**, in which code factoring is applied only to **bar.o** and **baz.o**.



Note Please note the following:

- Hand-written assembly code files and mixed assembly and C/C++ source files (that is, C or C++ source files with `#pragma startasm`, `asm` macros, etc.) cannot be code factored by default. For optimal results, do not mix both assembly and C/C++ code within the same source file. If code factoring of mixed assembly and C/C++ source files is necessary, see `#pragma ghs safeasm` and `#pragma ghs optasm` in “Green Hills Extension Pragma Directives” on page 683.
- When using the MULTI Debugger, you cannot set breakpoints on lines of source code that have been altered or removed by code factoring. These lines of code display a red breakdot instead of a green breakdot. Switch to interlaced source and assembly mode, and then set breakpoints at the assembly level.
- To control code factoring in the linker, the compiler generates a number of `.ghsnote` assembler directives (which reside in the non-allocated notes section `.ghs_info`, and have no impact on final code size). You should not modify these directives or the contents of this section.

Run-Time Clear and Copy Tables

These three tables are contained within the `.secinfo` section.

- The *clear* table is bounded by the symbols `__ghsbinfo_clear` and `__ghseinfo_clear`.
- The *copy* table is bounded by the symbols `__ghsbinfo_copy` and `__ghseinfo_copy`.
- The *compressed copy* table is bounded by the symbols `__ghsbinfo_comcopy` and `__ghseinfo_comcopy`.

These tables each contain zero or more records detailing the action to be taken at startup.

By default, the Green Hills C run-time library **libstartup.a** automatically clears `.bss`, and copies ROM and decompressed CROM sections to RAM based on this table. The actual implementation of the above three routines for your CRT code can be found in `src/libstartup/ind crt0.c` and `src/libstartup/ind_uzip.c` in your Green Hills product and differs slightly for PIC/PID support on some targets.

The ax Librarian

Chapter 12

This Chapter Contains:

- Creating a Library from the Project Manager
- Creating a Library from the Compiler Driver
- Modifying Libraries
- Creating and Updating a Table of Contents

The librarian combines object files created by the assembler or linker into a library (or archive file). By convention, libraries have the **.a** extension. At link time, the linker can search through libraries for object files, and pull them in to provide definitions for undefined symbols.

Creating a Library from the Project Manager

To create a library from the Project Manager:

1. Select your Top Project, or the project to which you want to add the library.
2. Click the  button on the toolbar, or right-click the project and select **Add Item Into**.
3. In the **Select Item to Add** dialog, select **Library** and click the **Next** button.
4. Specify a directory and enter the name of the new library. Click the **Finish** button.
5. In the Project Manager, select your new library. Add source files and projects to the library by using the  button on the toolbar, or by right-clicking the library and clicking the **Add Item Into** menu item.

When you build your library (or any project that contains the library), the Builder compiles and assembles the source files, and then runs the librarian to create a library from the resulting object files. For more information, see “Linking with a Pre-Built Library” on page 37.

Creating a Library from the Compiler Driver

To create a library using the compiler driver, use the following syntax:

```
ccppc filename... -archive -o libname.a
```

where each *filename* is a source file or object file that you want to include in the library, and *libname.a* is the name of the library you want the librarian to create. The compiler driver compiles and assembles the source files, and then runs the librarian to create a library from the resulting object files.

If *libname.a* already exists, the librarian inserts the new object files into the existing library. If the existing library contains an object file that has the same

name as one of the new object files, the librarian overwrites the existing object file with the new one.

Use the **-update_archive** option to force the librarian to delete all leftover object files from the library. The builder uses this option when creating and updating libraries.

To create a *merged library* – a library that merges together several existing libraries and/or object files – use the following syntax:

```
ccppc filename... input_library.a... -merge_archive -o output_library.a
```

-merge_archive is similar to **-archive**, with the following differences:

- If *output_library.a* exists, it is replaced.
- The entire contents of each specified *input_library.a* is added to the library (libraries specified with **-Ifile** are ignored). The contents of each input debug library (**.dba**) are added to the output debug library.
- Any input archives are passed to the prelinker after **--**. Templates that are instantiated in the archive may be used, but may not be rebuilt.

-merge_archive is implemented by passing the **C** and **M** command modifiers to the archiver (see “Librarian Command Modifiers” on page 477).

Example 1. How to Create a Library

To create a library **libfoo.a** from the source file **foo.c**, enter:

```
ccppc foo.c -archive -o libfoo.a
```

To create a library **libhello.a** from the object files **hello.o** and **world.o**, enter:

```
ccppc hello.o world.o -archive -o libhello.a
```

Modifying Libraries

You can modify libraries (**.a** files) by invoking the **ax** librarian directly, as follows:

```
ax -command [modifier]... [-option]... libraryfile... [memberfile]...
```

where:

- *-command* is one of the librarian commands
- each *modifier* is a command modifier
- each *-option* is a librarian option
- *libraryfile* is the name of the file you want to modify, including the extension.
- each *memberfile* is the name of the file (usually an object file) in the library you want to perform the command on, including the extension.

The following tables list valid command letters, modifiers, and options.

Librarian Commands

This table lists each librarian command, including the modifiers that you can use with it. The librarian command modifiers are explained in the subsequent table, “Librarian Command Modifiers” on page 477. Enter the command and modifiers as one string without spaces.

-d [e] [S] [v] <i>libraryfile memberfile...</i>
Deletes each specified <i>memberfile</i> from the specified <i>libraryfile</i> .
-p [e] <i>libraryfile memberfile...</i>
Prints each specified <i>memberfile</i> from the specified <i>libraryfile</i> to standard output (if the members are object files, then the output might not be human-readable).
-r [c] [C] [e] [S] [n] [m] [M] [v] <i>libraryfile memberfile...</i>
Replaces each specified <i>memberfile</i> in the specified <i>libraryfile</i> . The new version of <i>memberfile</i> overwrites the existing version. If <i>memberfile</i> does not already exist in <i>libraryfile</i> , the librarian adds it. If <i>libraryfile</i> does not exist, the librarian creates it. We recommend that you create libraries with the compiler driver and not with the librarian, because only the compiler driver generates the .dba files necessary for subsequent debugging.
-t [e] [s] [v] <i>libraryfile...</i>
Lists the names of the files contained in each specified <i>libraryfile</i> . Use the v option to list names, file sizes, and the dates that the members were last modified.
-x [e] [v] [o] <i>libraryfile memberfile...</i>
Extracts each specified <i>memberfile</i> from the specified <i>libraryfile</i> into the current directory. The librarian does not alter <i>libraryfile</i> . If you only specify <i>libraryfile</i> , the librarian extracts all member files.

Librarian Command Modifiers

These modifiers can be appended to the librarian command letters to give you greater control over operations on library files. See “Librarian Commands” on page 476 to determine which modifiers are available with each command letter.

c	Suppresses warnings when creating a library that did not exist.
C	Replaces the existing library.
e	Prefixes messages with <code>ERROR</code> or <code>WARNING</code> . Equivalent to the compiler driver option <code>-prefixed_msgs</code> .
n m M	These options control how the librarian handles input files that are libraries. They are mutually exclusive. • n — [default] The librarian treats all input files as complete units and does not consider that the input file might be a collection of files. Each input library is placed in the output library as one complete unit. • m — The librarian places each member of each input library into the output library as a separate file. • M — Similar to m, but the names of the files are mangled by prepending the base name of the archive to each file. The full path and file extension are not used in this mangling scheme. This modifier reduces the likelihood that the archive will contain two files with the same name.
o	Keeps original timestamps when extracting files from a library.
s	Regenerates the table of contents; use with the <code>-t</code> command.
S	Prevents the creation of a table of contents.
v	Prints verbose messages when executing a command.

Librarian Options

The following options are recognized on the command line before the name of the library:

-display_toc
Used with the t command letter to display the table of contents and the offset of each member file within the archive.
-pecoff
When generating the table of contents, this option also generates a Microsoft Windows compatible table of contents.
-stderr=errfile
Redirects error output of the librarian to the named file.
-tmp=dir
Specifies the temporary directory.

Examples

To add a new object file **foo.o** to an existing library **lib.a**, enter:

```
ax -r lib.a foo.o
```

To delete object files **foo.o** and **bar.o** from library **lib.a**, enter:

```
ax -d lib.a foo.o bar.o
```

To extract object file **foo.o** from library **lib.a** and rename it to **newfoo.o**, enter:

```
ax -p lib.a foo.o > newfoo.o
```

To print the table of contents of library **lib.a** in verbose mode, enter:

```
ax -tv lib.a
```

Creating and Updating a Table of Contents

The librarian creates a table of contents, by default, whenever it creates or modifies a library. This improves the linker's performance, because the linker does not have to create the table of contents every time it reads a library at link

time. The librarian stores the table of contents in a hidden member file named / (slash), which is always the first file in the library.

To subsequently create or update a table of contents without altering the library, use the **-t** command with the **s** option. For example, if you want to create a table of contents for **lib.a**, enter:

```
ax -ts lib.a
```


Utility Programs

Chapter 13

This Chapter Contains:

- The gaddr2line Utility Program
- The gasmlist Utility Program
- The gbin2c Utility Program
- The gbincmp Utility Program
- The gbldconvert Utility Program
- The gbuild Utility Program
- The gcolor Utility Program
- The gcompare Utility Program
- The gdump Utility Program
- The gfile Utility Program
- The gfunsize Utility Program
- The ghide Utility Program
- The gmemfile Utility Program
- The gnm Utility Program
- The gpjmodify Utility Program
- The grun Utility Program
- The gsize Utility Program
- The gsrec Utility Program
- The gstack Utility Program
- The gstrip Utility Program
- The gversion Utility Program
- The gwhat Utility Program

The MULTI IDE includes many useful command line utility programs, including functional replacements for the standard Linux/Solaris utilities **dump**, **file**, **hide**, **nm**, **size**, **strip**, and **what**. All utility programs work with files generated by any Green Hills development tools.

- **gaddr2line**: Similar to the Linux/Solaris **addr2line** program. Maps addresses to filenames and line numbers.
- **gasmlist**: Generates interlaced assembly and source output (object files only).
- **gbin2c**: Converts binary files into C array definitions.
- **gbincmp**: Compares two binary files (single object files or executables only).
- **gbldconvert**: Converts old-style build files (**.bld**) to new-style Green Hills project files (**.gpj**).
- **gbuild**: A command line interface to the MULTI Builder.
- **gcolor**: Takes text input and colors it using ANSI escape sequences.
- **gcompare**: Compares space or time performance.
- **gdump**: Similar to the Linux/Solaris **dump** program. Dumps or disassembles a file.
- **gfile**: Similar to the Linux/Solaris **file** program. Describes the file type.
- **gfunsize**: Returns the code size of a function.
- **ghide**: Similar to the Linux/Solaris **hide** program. Hides global symbols in an object file.
- **gmemfile**: Converts an executable file to a binary image suitable for loading.
- **gnm**: Similar to the Linux/Solaris **nm** program. Displays file information.
- **gpjmodify**: Performs command line editing of new-style **.gpj** project files.
- **grun**: (for executable files) Runs a server in batch mode.
- **gsize**: Similar to the Linux/Solaris **size** program. Displays section sizes.
- **gsrec**: Converts an executable file to Motorola S-Record format, Intel hexadecimal, or Tektronix hexadecimal format file.
- **gstack**: (for executable files) Computes stack size for each task.
- **gstrip**: Similar to the Linux/Solaris **strip** program. Removes symbol or debugging information from an executable.

- **gversion**: (for executable files) Returns version date and time information.
- **gwhat**: Similar to the Linux/Solaris **what** program. Reports or updates version information.

The gaddr2line Utility Program

The **gaddr2line** utility program displays the source filename and line number for instruction addresses.

The syntax for this utility is as follows:

gaddr2line [*options*] *addresses*

where *addresses* is a whitespace-separated list of instruction addresses and *options* are listed in the following table:

-f	Display function names along with filename and line number.
-e filename	Specifies the executable filename. The default is a.out .
-r	Display procedure relative line numbers instead of file relative line numbers.
-s	Display just the base filename, instead of the full path and filename.

The **gaddr2line** utility looks up each specified address in the debug information for the program and prints the source filename and line number that correspond to the address. The information is displayed as `filename:line number`. If there is no debug information, or the address does not correspond to a valid source line, then `?:0` is displayed. When the **-f** option is passed, the function name is displayed on the line preceding the source line information, and `??` is displayed for unknown functions. An example is shown below:

```
gaddr2line -e hello.out -f 0x10110 0x10114 0x99999999
main
hello.c:2
main
hello.c:4
??
?:0
```

The gasmlist Utility Program

The **gasmlist** utility program displays interlaced assembly and source for the specified object files.

The syntax for this utility is as follows:

gasmlist [*options*] *files*

where *files* is a whitespace-separated list of files and *options* are listed in the following table:

--all --all_with_asm Controls which files have source lines printed: • --all prints source lines for all files • --all_with_asm prints source lines for all files that produce assembly or, if --src_then_asm is specified, for all files. The default is to restrict output to the object file's base source file (so only source lines from the base source file are printed).
--file_num --proc_num Restricts identification of output lines to one of the following: • file line numbers • procedure line numbers The default is to print both procedure and file line numbers.
--file_offset Calculates offsets from the start of the present object file. The default is to calculate them from the start of the present section.
--help Prints usage information.
--hide_all_labels Suppresses the printing of detailed label information.
--hide_mangled Suppresses the printing of mangled versions of labels. The demangled versions are always printed.

-I *directory*

Specifies an additional *directory* to search for source files.

The following directories are searched for source files in this order:

- the relative or absolute path specified in the object file
- directories specified using **-I *directory***
- the user's path

The current directory is not searched unless it is specified in one of these categories.

--max_header_lines *n*

Specifies the maximum number *n* of header source lines to be printed.

The default value for *n* is 4. Setting *n* to 0 instructs **gasmlist** to print all associated header file lines.

--no_prefix_file

Suppresses the printing of the filename at the beginning of each source line.

--omit_headings

Suppresses the printing of headings to separate files and sections.

--proc_num_on_left

When both file and procedure line numbers are enabled, prints the procedure numbers on the left. The default is to print them on the right.

--root_remap *rt dir*

Re-maps the original path *rt* to the new path *dir*. For example to map **a/b/c** to **d/c**, pass:

```
--root_remap a/b d
```

--show_func_labels

Prints line offsets with the label they are offset from. For instance, `+ 0x4` would instead be printed `label + 0x4`.

--src_then_asm

Orients the output primarily towards source code. The default is to emphasize assembly.

The gbin2c Utility Program

The **gbin2c** utility converts a group of binary files into a set of array definitions in a source file suitable for inclusion in C or C++ projects. You can control the byte by byte conversion of each binary file into a char array by passing command line arguments.

The syntax for this utility is:

gbin2c [*global_options*] [-input] *file1* [*file1_local_options*]....

Command line options for **gbin2c** are divided into three categories.

- *Global options* — Apply to each binary file that you pass to **gbin2c** for conversion. You can specify global options anywhere on the command line.
- *Module inclusion options* — Specify the binary files that you want to convert.
- *Local options* — Apply only to the binary file specified by the nearest preceding module inclusion option.

Global Options

-allnoconst Removes const from a data array type for all modules and their references in the module list.
-compat Runs gbin2c in gbin2src compatibility mode. Implies: • the -list global option • the -static local option for each binary file. If an additional options file is not specified on the command line (with -i), this option also instructs gbin2c to read standard input for additional options in lieu of an additional options file.
-crc Outputs a CRC value for each converted binary file, using the polynomial 0x10211021. Sample source code for verifying checksums is included in the src/libstartup/cksum.c file of your installation. The declaration of the CRC variable for the array named <i>name</i> is: <pre>const unsigned int name_crc;</pre>

-cxx

Replaces `const` with `extern const`. Use this option if you will be including the output in a C++ project.

-ghsentry *tablename*

Outputs an entry for the linker generated table *tablename* for each converted module. Use this option as an alternative to `-list`. For more information, see “Linker Generated Tables” on page 465.

-help

Displays basic usage information.

-i *file*

Reads *file* as a list of additional options.

The contents of *file* are parsed as additional options. Extra whitespace and newline characters are ignored. Options listed in the file have the same effect as options on the command line with the following exceptions:

- The options file is not preprocessed by the user’s shell so no special treatment, such as wildcard substitution, is performed.
- The meaning of the `-o` option in the additional options file context differs from its meaning in the command line context. In the options file, `-o name` means “refer to the current module as *name* in the module list, and set the array name for the current module to *name_data*”. In essence, the `-o` option is an alias for the `-output` option when it occurs on the command line and the `-module_name` and `-array_name` options when it occurs in the options file.

Note that only one additional options file can be specified using the `-i` option.

-list

Outputs a list of converted modules.

The default definition for the type of each list element is:

```
struct static_module_info {  
    const char * const name;  
    const char * const data;  
    const struct static_module_info * const next;  
};
```

The default declaration of the module list is:

```
const struct static_module_info __static_modules[];
```

-list_name *name*

Sets the name of the module list to *name*.

-list_type <i>name</i>
Sets the name of the module list element struct type to <i>name</i> .
-nullterm
Terminates each data array with a null character.
-o <i>file</i> -output <i>file</i>
Writes output to <i>file</i> . See -i , above, for the -o option's alternate meaning in an options file. If neither of these options is passed, then output is written to <code>stdout</code> .
output <i>file</i>
Writes output to <i>file</i> .
-size
Outputs the length of each array written. The declaration of the size variable for the array named <i>name</i> is: <pre>const unsigned int <i>name_size</i>;</pre>
-size_array
Outputs an array containing the lengths of all data arrays written when -list is specified
-size_field
Adds a <code>size</code> field to the module list struct type.
-string
Formats output in string literal form instead of comma-separated hex values.

Module Inclusion Options

[-input] *file*

Adds *file* to the list of binary files to convert.

An array is defined for each converted binary file. The default declaration for the array generated from file *file* (stripped of all path information, and with any non-alphanumeric character replaced with underscores) is:

```
const char file_data[];
```

The nature of the array can be controlled via the *local options*.

Note: Non-alphanumeric special characters in the name of *file* are replaced with underscores.

Local Options

These options modify settings for the most recently included module:

-alignment *n*

Sets the alignment of an array for the current module.

-array_name *name*

Sets the array name for the current module to *name*.

By default the array name for a file is the base name of the file with all non-alphanumeric characters replaced with underscores.

-limit *n*

Outputs a maximum of *n* bytes of the current module.

-module_name *name*

Refers to the current module as *name* in the module list.

-noconst

Removes `const` from the type of the array for the current module.

-signed**-unsigned**

Adds `signed` or `unsigned` to the type of the array for the current module.

-skip *n*

Skips the first *n* bytes of the current module.

-static

Adds `static` to the type of the array for the current module.

-volatile

Adds `volatile` to the type of array for the current module.

Example 1. Using gbin2c

```
gbin2c mod.o
```

The contents of **mod.o** are converted to a `const char` array named `mod_o_data` and printed on `stdout`. The output looks similar to the following:

```
/**
 * This file was automatically generated by gbin2c v1.08
 * Green Hills Software, Inc. [ http://www.ghs.com ]
 * Using command line: gbin2c mod.o
 * Date generated: Wed Jul  9 16:04:18 2003
 **/

/* mod.o: */

const char mod_o_data[] = {
    0x7F, 0x45, etc.
};
```

Example 2. Using the -crc and -size Options

```
gbin2c -crc -size mod.o
```

In addition to the output from example 1, size and CRC information for **mod.o** are displayed:

```
/**
 * This file was automatically generated by gbin2c v1.08
 * Green Hills Software, Inc. [ http://www.ghs.com ]
 * Using command line: gbin2c -crc -size mod.o
 * Date generated: Wed Jul  9 16:09:48 2003
 * Note: CRC of input file and this array can be found at end of file
 **/

/* mod.o: */

const char mod_o_data[] = {
    0x7F, 0x45, etc.
};

const unsigned int mod_o_data_size = 2620;

const unsigned int mod_o_data_crc = 0xD91D5642;
/* CRC of input file = 0xD91D5642 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */
```

Example 3. Using the static Keyword

```
gbin2c -crc -size mod.o -static -skip 64 -limit 4
```

In this example, the `static` keyword is added to the type of `mod_o_data` and only bytes 64-67 (starting counting at 0) of **mod.o** are output:

```
/**
 * This file was automatically generated by gbin2c v1.08
 * Green Hills Software, Inc. [ http://www.ghs.com ]
 * Using command line: gbin2c -crc -size mod.o -static -skip 64 -limit 4
 * Date generated: Wed Jul  9 16:16:19 2003
 * Note: CRC of input file and this array can be found at end of file
 **/

/* mod.o:  skipped 64 bytes and limited to 4 bytes */

static const char mod_o_data[] = {
    0x65, 0x2F, 0x73, 0x73
};

const unsigned int mod_o_data_size = 4;

const unsigned int mod_o_data_crc = 0x358E11C6;
/* CRC of input file = 0xD91D5642 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */
```

Example 4. Converting Multiple Files

```
gbin2c -crc mod.o -static foo -signed -size -array_name bar
```

In this example, both **mod.o** and **foo** are converted. Notice that the **-crc** and **-size** global options apply to both files while **-static** and **-signed** only apply to **mod.o** and **foo** respectively. This example also demonstrates the use of the **-array_name** option and its effect on the size and CRC variable names.

```
/**
 * This file was automatically generated by gbin2c v1.08
 * Green Hills Software, Inc. [ http://www.ghs.com ]
 * Using command line:
 *   gbin2c -crc mod.o -static foo -signed -size -array_name bar
 * Date generated: Wed Jul  9 16:28:19 2003
 * Note: CRC of input file and this array can be found at end of file
 **/

/* foo: */

const signed char bar[] = {
    0x66, 0x64, 0x61, 0x73, 0x0A
};

const unsigned int bar_size = 5;

const unsigned int bar_crc = 0x11A9EBE1;
/* CRC of input file = 0x11A9EBE1 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */

/* mod.o: */

static const char mod_o_data[] = {
    0x7F, 0x45, etc.
};

const unsigned int mod_o_data_size = 2620;

const unsigned int mod_o_data_crc = 0xD91D5642;
/* CRC of input file = 0xD91D5642 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */
```

Example 5. Outputting the Module List

```
gbin2c mod.o foo -module_name bar -list
```

In this example the module list is output and the module name for **foo** is changed to **bar**.

```
/**
 * This file was automatically generated by gbin2c v1.08
 * Green Hills Software, Inc. [ http://www.ghs.com ]
 * Using command line: gbin2c mod.o foo -module_name bar -list
 * Date generated: Wed Jul  9 16:30:48 2003
 **/

struct static_module_info {
    const char * const name;
    const char * const data;
    const struct static_module_info * const next;
};

/* foo: */

const char foo_data[] = {
    0x66, 0x64, 0x61, 0x73, 0x0A
};

/* mod.o: */

const char mod_o_data[] = {
    0x7F, 0x45, etc.
};

const struct static_module_info __static_modules[] = {
    {
        "bar",
        (const char *)&foo_data,
        &__static_modules[1]
    },
    {
        "mod.o",
        (const char *)&mod_o_data,
        0
    }
};
```

Example 6. Including an Additional Options File

```
gbin2c -i modules -list -crc
```

In this example, the **-i** option is used to include an additional options file called **modules**, which contains the following text:

```
mod.o
foo -o bar
```

The **-o** option in **modules** specifies that the module name for **foo** should be **bar** and that its array name should be **bar_data**.

```
/**
 * This file was automatically generated by gbin2c v1.08
 * Green Hills Software, Inc. [ http://www.ghs.com ]
 * Using command line: gbin2c -i modules -list -crc
 * Additional options from file: mod.o foo -o bar
 * Date generated: Wed Jul  9 16:38:51 2003
 * Note: CRC of input file and this array can be found at end of file
 */

struct static_module_info {
    const char * const name;
    const char * const data;
    const struct static_module_info * const next;
};

/* foo: */

const char bar_data[] = {
    0x66, 0x64, 0x61, 0x73, 0x0A
};

const unsigned int bar_data_crc = 0x11A9EBE1;
/* CRC of input file = 0x11A9EBE1 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */

/* mod.o: */

const char mod_o_data[] = {
    0x7F, 0x45, etc.
};

const unsigned int mod_o_data_crc = 0xD91D5642;
/* CRC of input file = 0xD91D5642 */
/* Note that file CRC may not match array CRC if -skip or -limit were used */

const struct static_module_info __static_modules[] = {
    {
        "bar",
        (const char *)&bar_data,
        &__static_modules[1]
    },
    {
        "mod.o",
        (const char *)&mod_o_data,
        0
    }
};
```

The **gbincmp** Utility Program

The **gbincmp** utility program compares two binary input files. The two input files can be ELF format object files or executable files.

If the sections in the two files are identical, **gbincmp** exits with no output. If they differ, it prints the first difference, unless you specify the **-continue** option.

The **gbincmp** utility program can compare any of the following:

- Contents of sections loaded on the target (target sections)
- Contents of sections not loaded on the target (host sections)
- Section headers
- Relocation tables
- String tables
- Symbol tables
- DWARF debug information sections

The syntax for this utility is as follows:

gbincmp [*options*] *file1 file2*

where *file1* and *file2* are the names of the two files to be compared.

The **gbincmp** *options* are:

-all
Compares all sections.
-continue
Instructs gbincmp not to stop at the first difference.
-debug -nodebug
Enables or disables comparison of the <code>.debug</code> (DWARF) and <code>.debug_info</code> (DWARF2) sections.
-help
Displays information about all options.

-host -nohost Enables or disables comparison of the contents of host sections.
-line -noline Enables or disables comparison of the line number information sections <code>.line</code> (DWARF) and <code>.debug_line</code> (DWARF2).
-normalize Ignores differences in encoded timestamps.
-prefixed_msgs -no_prefixed_msgs Enables or disables the insertion of the words <code>WARNING</code> and <code>ERROR</code> before every warning and error message.
-reloc -noreloc Enables or disables comparison of relocation information.
-section=sections -nosection=sections Enables or disables comparison of the sections named in the comma-separated <i>sections</i> list. For example, <code>-section=text,data</code> (ELF only).
-string -nostring Enables or disables comparison of string table information.
-symbol -nosymbol Enables or disables comparison of symbol table information.
-target -notarget Enables or disables comparison of the contents of target sections.
-v Verbose output. Shows what files and sections are compared.

The gbldconvert Utility Program

The **gbldconvert** utility program converts legacy build files (**.bld**) to Green Hills project files (**.gpj**). For more information, see *MULTI Builder: Migrating Legacy Projects*.

The gbuild Utility Program

The **gbuild** utility program provides a command line interface to build projects created with the MULTI Project Manager (see Chapter 2, “The MULTI Project Manager”). This utility can be used to integrate building your project into a script.

The syntax for this utility is as follows:

gbuild [*option*]... [*file*]...

where the optional *file* parameter can be:

- one or more project files (**.gpj**)
- one or more source or object files
- one or more executables

If you do not specify *file*, **gbuild** builds the file specified by the **-top** option. If you do not specify **-top**, **gbuild** tries to build **default.gpj** in the current directory.

The **gbuild** *options* are:

-#	Displays status messages and commands but does not perform any actions. This option is a shortcut for -info -commands .
-all	Rebuilds all files, even if they are up to date.
-allinfo	Displays the status messages that you would encounter if you rebuilt all files. This option does not actually build your files or perform the actions described in the messages. This option is a shortcut for -all -info .
-clean	Deletes all previous output files without building.
-cleanfirst	Deletes all existing output files and then initiates a build.
-commands	Displays commands before they are executed.

-Dmacro_name [=value]

Defines *macro_name* as *value* for use in project (**.gpj**) files and customization (**.bod**) files.

This option does not define a preprocessor symbol.

-find filename

Displays the first build path to the file *filename*. The build path specifies a route through the build structure to reach *filename*.

-findall filename

Displays all build paths to the file *filename*. The build paths specify routes through the build structure to reach *filename*.

--help**-help****-h**

Displays usage information for basic **gbuild options**.

-Help**-H**

Displays usage information for all **gbuild options**.

-ignore

Continues building, even if compile or link errors occur.

-ignore_parse_errors

Continues building even if errors occur while parsing project files. Some critical errors cannot be ignored and will still halt the build.

-info

Displays status messages but does not perform the actions described in the messages.

-leave_output

Deletes only intermediate output files. This option is only available with **-clean** or **-cleanfirst**.

-link

Forces linking. The default behavior is to re-link the executable only if any of its components have changed.

-list

Lists all filenames. This option does not perform any build actions.

-list_depends

Lists known dependencies. You must build your project before passing this option. This option does not perform any build actions.

-list_options

Lists options specified in the **.gpj** file. This option does not perform any build actions.

-lockout

Uses a lock file to prevent simultaneous builds of the same program or library.

-lockoutretry

Waits to obtain a lock file. This option implies **-lockout**.

-nested_commands

Displays full commands, including internal commands, before they are executed.

-nice

Executes the build with low priority.

-nochecklibs

Disables Implicit Dependency Analysis. For more information, see “Implicit Dependency Analysis” on page 505.

-nolink

Generates object files but does not link them.

-noparallel

Specifies that **gbuild** cannot run parallel processes for the build. The build will be single threaded.

-noprogess

Suppresses progress messages.

-noreasons

Restricts the information given in progress messages.

For example, the message:

```
Compiling hello.c because hello.obj does not exist
```

is reduced to:

```
Compiling hello.c
```

-parallel[=*n*]

Specifies the number of processes that **gbuild** can run in parallel. The default value for *n* is 2.

-parallel=0 is equivalent to -noparallel.

-path *list_of_project_files file*

Specifies a build path of project (**.gpj**) files to follow when building the file *file*. The build path specifies a route through the build structure to reach *file*. When using **-path**, you cannot specify any file arguments to **gbuild** other than the one specified for **-path**. This option is helpful when one file is utilized in multiple paths in your build tree, and you want to specify which path to use for building.

-preprocess

Only performs preprocessing on a project, generating **.i** files from applicable source.

-quiet**-silent**

Prevents the utility from outputting any warning or error messages.

-skip=*target*

Skips the specified build target *target* when building. Here, *target* is the output name associated with a build file.

-statistics

Displays comprehensive timing information.

-stderr=*filename*

Redirects stderr output (status and error messages) to *filename*.

-strict

Aborts **gbuild** if **gbuild** is unable to open a **.gpj** file in your build tree or if you specify an unrecognized option on the command line. Without this option, these problems only generate warning messages.

-time

Displays basic timing information.

-top *filename.gpj*

Specifies the Top Project file. By default, **gbuild** uses **default.gpj** in the current directory.

This option allows you to specify a path to a project file in another directory.

-unix

Displays directory slashes as **/**.

-verbose

Prints additional information during the build.

-within *intermediate.gpj*

Restricts searching for the specified *file* arguments to children of *intermediate.gpj*.

This option is an alternative for **-path**; it allows you to specify multiple files, while restricting the search for those files to a sub-tree of your project. This option is helpful when there are multiple files with identical names in multiple sub-trees.

Using gbuild

The following examples take place in a directory, **MyProjects/Project1**, which contains a Top Project file called **default.gpj**, and a number of levels of subprojects.

Example 7. Building the Entire Project

To build the entire project, enter the following:

```
gbuild
```

If you do not pass any arguments, **gbuild** attempts to process **default.gpj** in the current directory.

Example 8. Building Part of Your Project

To build a particular executable, you must specify either its project file or the name of the executable itself.

For example, to build **print_utility.gpj**, a project file that generates an executable called **print_utility**, enter the following:

```
gbuild print_utility.gpj
```

or:

```
gbuild print_utility
```

Example 9. Building Individual Object Files

To build individual object files, you must specify either the initial source files or the object files that they will be compiled into.

For example, to build the source files **foo.c**, **bar.s**, and **baz.cxx** into object files, enter the following:

```
gbuild foo.c bar.s baz.cxx
```

or:

```
gbuild foo.o bar.o baz.o
```

Example 10. Finding Project Files

As your project grows in size, it might become difficult to find individual files, particularly if you are using several levels of subprojects. To search for the project file **print_utility.gpj**, enter the following:

```
gbuild -find print_utility.gpj
```

This command prints the position of **print_utility.gpj** in the project tree and the path to it from the current directory. The output resembles the following:

```
default.gpj utilities.gpj utilities/print_utility/print_utility.gpj
```

In this case, **print_utility.gpj** is a subproject of **utilities.gpj**, which is a subproject of **default.gpj**. The actual file can be found at **utilities/print_utility/print_utility.gpj**.

Example 11. Building One of Multiple Projects with the Same Name

If your project contains identically named sub-projects, use the **-path** or **-within** options to specify which project to build. For example, if your project structure is:

```
default.gpj
  target_one.gpj
    foo.gpj
    lib.gpj
  target_two.gpj
    foo.gpj
    lib.gpj
```

- To build the **lib.gpj** project in **target_two.gpj**, use the following command:

```
gbuild -path default.gpj target_two.gpj lib.gpj
```

The **-path** option only allows you to specify one item to build.

- To build both the **lib.gpj** and **foo.gpj** projects in **target_two.gpj**, use the following command:

```
gbuild -within target_two.gpj foo.gpj lib.gpj
```

Example 12. Running gbuild in a Project Subdirectory

When building a portion of your project, **gbuild** looks for a **default.gpj** in the current directory to ensure that any build options set in the upper portions of the project structure are inherited appropriately. If your Top Project is not named **default.gpj** or does not exist in the current directory, you must specify its location with the **-top** option.

For example, if you have navigated to **utilities/print_utility/**, and intend to build **print_utility.gpj**, you should enter:

```
gbuild -top ../../default.gpj print_utility.gpj
```

Implicit Dependency Analysis

Implicit Dependency Analysis (IDA) attempts to locate dependencies for a program that are not explicitly specified. Explicitly specified dependencies are listed as children of a given program. Programs might depend on more than just

what is listed, however, the remaining dependencies are considered implicit. There are 2 specific kinds of implicit dependencies IDA searches for:

1. Additional libraries referenced by **Prebuilt Library** objects within a project.
2. Additional libraries or objects pulled in by options (for example, **-l**, **-kernel**, **-linker.args**), or declared by using **:depends**.

The first type of IDA is performed as soon as the builder encounters a **Prebuilt Library** object. The second type of IDA is performed after all children of a program have been built, and before the link phase begins.

gbuild attempts to locate the build item that produces additional dependencies found via IDA, and builds the dependencies automatically if they are not current. Dependencies that **gbuild** cannot build (for instance, because they are system libraries) are ignored.

When searching for dependencies, **gbuild** only searches within Projects and Subprojects that contain items that can be built. **gbuild** does not search within items that can be built; thus, objects that can be found by IDA must be located near the top-level of the build tree. Lastly, IDA does not search for dependencies within projects that cannot be reached from the current top-level build file.

An implicit dependency can be defined after an item that references it. As long as it meets the preceding requirements, IDA will still find and build the implicit dependency. **gbuild** does not perform IDA during the clean phase of a build. Thus, implicit dependencies are not cleaned. You must clean the dependencies separately, or clean a parent project.

IDA is enabled by default, and can be disabled by passing the **-nochecklibs** option to **gbuild** on either the command line or via the Project Manager.

The gcolor Utility Program

The **gcolor** program takes text input, colors it using ANSI escape sequences, and outputs it to `stdout`.

The syntax for this utility is as follows:

gcolor [*options*] [*filenames*]...

where *filenames* is an optional list of one or more whitespace-delimited input filenames. If no files are specified, `stdin` is used.

The *options* for **gcolor** are as follows:

-f	Forces coloring even when output is redirected.
-help	Shows the list of gcolor options.
-r file	Reads the coloring rules from <i>file</i> . If this option is not passed, the default rules contained within defaults/colors/build.rul are used.

The **gcolor** utility reads in a set of rules from a rules file to determine how the input should be colored. If a line from the input cannot be identified as conforming to any rule, then no coloring is performed.

There are five types that can be defined using rules that are recognized by **gcolor**. Each one can output a specific foreground, background, and attribute.

Rule	Default Coloring
info	Foreground = Bold, Default
warning	Foreground = Bold, Magenta
error	Foreground = Bold, Red
before	Foreground = Normal, Default
after	Foreground = Bold, Default



Note The types `before` and `after` are used for coloring output from a `diff`, for example when comparing two files.

Color Configuration Files

The color configuration for each of the types can be controlled by creating a **colors.ini** file and saving it in your user configuration directory.

The syntax of a color configuration file is as follows:

```
[type]
attribute = value
foreground = value
background = value
```

where *type* and *value* are listed in the following tables:

Types
info
warning
error
before
after

Colors	
Value	Background or Foreground
-1	Default
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White

Attributes	
Value	Attribute
-1	Default
0	None
1	Bold
4	Underscore
5	Blink
7	Reverse
8	Concealed

Comment lines must be prefixed with a '#' character as the first non-whitespace character.

For example, to make `info` display in normal white text on a blue background, you would use the following:

```
[info]
attribute = 0
foreground = 7
background = 4
```



Note If you do not specify coloring for a type, that type is colored in the default manner. If you do not specify *values* for attribute, foreground, or background, the default values are used.

Rules Files

The default set of rules is located in **defaults/colors/build.rul**. These correspond to the output from the builder and any of the programs that it invokes (the compiler, assembler, linker, etc.).

Alternatively, you can specify your own set of rules to use with the **-r file** option. You must create a rules file (*file.rul*) and save it in the following location:

- Site-wide rules should be saved in the **config/colors** directory.
- User-specific rules should be saved in your **.ghs/colors** directory.

Writing a Rules File

Rules are described using regular expressions. To add a rule to a rules file, first you must give the type of the rule, followed by a colon, and then the desired regular expression. The regular expression begins with the first non-whitespace character following the colon.

For example, if you wanted to define directories in a standard Linux/Solaris-style directory listing as `info`, you can use a simple rule such as:

```
info: /$
```

There are two special rule types: `none` and `previous`. The `none` type causes matching text to be printed without color. This is useful if you want to prevent the coloring of specific lines that would otherwise match against a later rule. For example, the rules:

```
none: ^foobar  
info: ^foo
```

would color lines beginning with `foo`, except those beginning with `foobar`.

The `previous` type causes matching text to be colored the same as the previous line of text. This is useful to ensure that continuation lines are colored the same as their parents.

Comment lines must be prefixed with a '#' character as the first non-whitespace character.

Example 13. Coloring a File

To color file **foo**, using your own set of rules contained within **myrules.rul**:

```
gcolor -r myrules.rul foo
```

To color builder output using the standard rules, enter (for **sh** shells):

```
gbuild default.gpj 2>&1 | gcolor
```

Note that the syntax for **cs** shells is different. If you are using a **cs** shell, enter:

```
gbuild default.gpj |& gcolor
```



Note Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyrighted by the University of Cambridge, England. It can be downloaded from the University of Cambridge FTP server (<ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>).

The gcompare Utility Program

The **gcompare** utility program compares the code size of matching functions in two input files and produces a report. Input files can be ASCII text files, object files, object file libraries, or executable files. If an input file is a text file, it consists of lines in the following format:

```
name1  number1
name2  number2
...
```

Each name/number pair is on its own line, separated by blanks or tabs.

If either input file is an object file, an object library file, or an executable file, **gcompare** automatically runs **gfunsize -gcompare** (or another command specified by the **-x** option) to produce an input text file to compare.

You can mix all input file types with no restrictions, including files for different target CPUs.

The **gcompare** utility reads both input files. For any name that does not exist in both files, **gcompare** prints a warning, unless **-w** is specified. For each name that exists in both files, **gcompare** compares the old and new numbers. If the new number is worse (larger is worse unless **-i** is specified), **gcompare** writes a line to the output report file containing the name, new number, old number, and percentage by which the new number is worse.

For example, if the size of the function `main` has increased from 28 bytes in **a.out** to 32 bytes in **b.out**, then `gcompare a.out b.out` will produce:

```
main          32      28   -14%
```

The syntax for this utility is as follows:

gcompare [*options*] *oldfile newfile*

where *oldfile* is the baseline input file and *newfile* is the input file for which deviations (improvements or degradations) are reported.

The **gcompare** *options* are:

-1
Produces a report with comparisons sorted from worst to best. All comparisons (including those in which <i>newfile</i> is no larger than <i>oldfile</i>) are shown. In addition, a summary is provided at the top, showing the aggregate size comparison between <i>newfile</i> and <i>oldfile</i> .
-a
Prints comparisons even when items exist in only one file.
-help
Displays information about all options.
-i
With -i , larger is better. Without -i , smaller is better.
-ignore name
Suppresses comparisons of symbol <i>name</i> .
-L
Formats the report in 132-column (landscape) mode. The default is 80-column (portrait) mode.
-min=n
Suppresses comparisons where the difference between the compared items is less than <i>n</i> bytes.
-r
Produces a 2-column report with comparisons sorted both from worst to best and from best to worst.
-v
Verbose mode, similar to -1 , but with additional diagnostics.
-w
Suppresses warnings.
-width=n
Sets width (<i>n</i>) to any value greater than 32.

-x cmd

Specifies the command to execute on an input file recognized as an object, object library, or executable file. The default **-x** command is:

```
-x "gfunsize -gcompare -all"
```

-z

Does not show cases where files are the same.

If the **-r**, **-l**, or **-v** options are used, all comparisons are shown, regardless of the result of the comparison. With **-r**, two reports are shown side by side in two columns. The left column is sorted from best to worst, and the right column is sorted from worst to best.

Sample **-r** output:

linpack.OS.o					linpack.O.o			
vs linpack.O.o					vs linpack.OS.o			
BETTER by:	3208	3476	7.71%		WORSE by:	3476	3208	8.35%
-----	-----	-----	-----		-----	-----	-----	-----
epsilon	20	20	0%		dgefa	484	356	-36%
main	912	928	2%		matgen	304	252	-21%
dmxpy	1188	1224	3%		dgesl	272	248	-10%
idamax	108	112	4%		dscal	52	48	-8%
daxpy	76	80	5%		daxpy	80	76	-5%
dscal	48	52	8%		idamax	112	108	-4%
dgesl	248	272	9%		dmxpy	1224	1188	-3%
matgen	252	304	17%		main	928	912	-2%
dgefa	356	484	26%		epsilon	20	20	0%

This output can require many columns. Use the **-L** option for a 132-column format. On Linux/Solaris, use the **lpr -L** command to print in landscape mode.

The gdump Utility Program

The **gdump** utility program performs a similar function to the Linux/Solaris **dump** program. It produces information about various types of files.

The syntax for this utility is as follows:

gdump [*options*] *filename*

where *filename* is the name of the input file, and *options* are listed in one of the following tables:

Debugging File Options

You can use the **gdump** utility to print symbolic debugging information from a **.dbo**, **.dba**, **.dlo**, **.dla**, or **.dnm** file. The options for **.dnm** files are different than those for the other debugging files.

-c	Performs internal consistency checks.
-C	Performs internal consistency checks, but ignores warnings.

-dinfo

Enables the display of particular information, where *info* is a letter from the lists below.

The following values for *info* are available for all debugging file formats:

- a — the attributes
- f — the file table
- h — the file header
- t — the typedef table

The following values for *info* are available for all debugging file formats except **.dnm**:

- c — static call information
- d — the #define table
- i — include reference information
- m — frames
- o — code trees
- p — the procedure table
- s — the symbol table
- u — the auxiliary table
- x — cross reference information

The following values for *info* are available for the **.dnm** file format only:

- b — linkage stubs
- d — **.dlo** files
- e — the global #define table
- i — the trace regions table
- k — link labels
- l — libraries
- s — sections
- u — **.dnm** updates

-help

Displays information about all options.

-v

Performs internal consistency checks only. Does not display symbols.

-V

Same as **-v**, but ignores warnings.

ELF File Options

The **gdump** *options* for ELF format files are as follows:

-asm
Produces text sections as pure assembly language (see also the -ytext option).
-dwarf
Produces DWARF information only.
-full
Produces everything except section contents (see also the -ysec option).
-help
Displays information about all options.
-integrity_checksum
Applies the checksum calculation method used by INTEGRITY when the -print_checksum or -verify_checksum options are specified.
-load
Displays ELF header summary.
-map
Displays section summary.
-N
Only displays information as indicated by the -y options.
-print_checksum
Prints the checksum for each appropriate section. If -verify_checksum is also specified, checksums are assumed to exist and -print_checksum prints them for those sections where the checksum is found to be correct. If the -verify_checksum option is not specified, checksums are assumed not to exist in the section, so they are calculated using all bytes in the section.
-raw
If the -ysec option is specified, this produces text sections in hexadecimal format instead of disassembly.
-sx
-nx
Enables or disables attempted shorter C++ demangling.
-sym
Uses symbol names instead of numbers in relocation output.

-v1**-v2**

Specifies display of DWARF Version 1 or 2. The DWARF version in use is detected automatically by **gdump**, and the appropriate option set automatically.

-verify_checksum

Instructs **gdump** that all non-empty, allocated sections have a 4-byte checksum generated by the Green Hills linker. Compares the content of each section against the existing checksum and, if they do not match, displays both.

-yinfo**-ninfo**

Enables or disables the display of particular information, where *info* is one of:

- **d**: DWARF debugging information
- **dynamic**: dynamic linkage information
- **f**: DWARF call frame information
- **g**: global offset table
- **h**: ELF header information
- **l**: DWARF line number information
- **m**: DWARF macro information
- **r**: relocation information
- **p**: procedure linkage table
- **s**: symbol table information
- **sec**: section contents
- **sh**: section header information
- **str**: string table information
- **text**: text section contents

COFF File Options

The **gdump** *options* for COFF files are as follows:

-ascii	Displays section contents in ASCII.
-aux	Displays auxiliary entries (in hex) with symbol entries.
-bigendian -littleendian	Forces either a big or little endian target.
-brief	Displays symbols only.
-c	Displays string table.
-cpu=<i>name</i>	Sets CPU type (<i>name</i>).
-d <i>num</i> [+d <i>num2</i>]	Displays section <i>num</i> or sections from <i>num</i> to <i>num2</i> .
-f	Displays file header.
-full	Displays everything.
-g	Displays global symbols.
-h	Displays section headers.
-helpcoff	Displays all options relating to COFF files.
-l	Displays line number information.
-map	Displays section summary.

-nx
Does not attempt to demangle C++ names.
-o
Displays optional header.
-r
Displays relocation information.
-raw
Produces section contents in raw form.
-s
Produces section contents.
-sx
Attempts shorter C++ demangling.
-sym
Uses symbol names instead of numbers in relocation output.
-t <i>num</i> [+t <i>num2</i>]
Displays symbol <i>num</i> or symbols from <i>num</i> to <i>num2</i> .
-ysec -ytext
Enables the display of section contents and/or text section contents.

BSD File Options

The **gdump** *options* for BSD format files are as follows:

-c
Displays section contents.
-h
Displays file header.
-helpbsd
Displays information about all BSD specific options.

-r

Displays relocation entries.

-s

Displays symbol table.

The gfile Utility Program

The **gfile** utility program performs a similar function to the Linux/Solaris **file** program. It displays the file type of each filename argument, and can also display additional information. For example, if a machine supports both big endian and little endian data ordering, then for an object file, object file library, or executable file, **gfile** displays the machine type and byte order. For unrecognized object files, **gfile** prints unknown machine type. In general, **gfile** handles BSD, ELF, and some COFF formats.

The syntax for this utility is as follows:

gfile *filenames...*

where *filenames* is a list of one or more filenames separated by whitespace.

If the current host system is a SPARC workstation running the Solaris 2.x operating system (which uses the ELF format for executable files), you will see the following behavior:

```
gfile /bin/od
/bin/od:      SPARC big endian executable (ELF)
```

The gfunsize Utility Program

The **gfunsize** utility program displays the code size of one or more named functions in an object file, library file, or executable. The code size of ELF functions is usually part of the ELF symbol information.

The syntax for this utility is as follows:

gfunsize [*options*] *filename*

where *filename* is the file to examine, and *options* are listed in the following table:

-addr
Displays the address of each function.
-file
Displays the filename before each function.
-func=<i>name</i>
Displays the code sizes of the specified function or functions.
-gcompare
Displays the output in a format suitable for input to the gcompare utility program.
-help
Displays information about all options.
-hex
Displays function code sizes and addresses in hexadecimal.
-nunderscores
Removes leading underscores from function names.
-sect=<i>name</i>
-sectnum=<i>n</i>
-sect =Displays functions in section <i>name</i> .
-sectnum =Displays functions in section number <i>n</i> .
These options are mutually exclusive.
-w
Suppresses warnings.

The **ghide** Utility Program

The **ghide** utility program modifies the symbol table of an existing object file by converting all global symbols to local symbols, except for a specified list of global symbols that remain global. The object file can be either relocatable or not relocatable, but it must contain a symbol table; otherwise **ghide** has no effect.

The syntax for this utility is as follows:

ghide [*options*] *retain_list* *object_file*

where *retain_list* is the name of an input file containing symbol names, separated by whitespace or newline characters, that are to be retained (not hidden), and *object_file* is the name of the existing object file to be modified by **ghide**.

The **ghide** *options* are:

-help

Displays information about ghide .

Using **ghide**

Suppose an embedded application system consists of a kernel and several application tasks, all developed independently. Some global functions in the kernel are for kernel use only. Other global kernel functions can be called from the application tasks.

First, the kernel is linked into a single file using the linker's **-r** option to retain relocation information. Then, **ghide** is run on the kernel using a retain list of the functions that should remain visible to the application tasks. Finally, the application tasks are linked with the kernel file, producing a complete executable file.

The use of **ghide** ensures that only the desired global symbols in the kernel are visible to the application tasks. This also prevents duplicate symbol errors in the linker if any application might have a global symbol with the same name as an internal kernel-only function.

The gmemfile Utility Program

The **gmemfile** utility program creates a binary memory image of a fully linked executable. This memory image is suitable for downloading to target memory. Use the `-memory` driver option to automatically invoke `gmemfile`. For more information about `-memory`, see “**Generate Additional Output**” on page 244.

The syntax for this utility is as follows:

gmemfile *executable* [*options*]

where *executable* is the name of the executable from which to generate the memory image, and *options* are listed in the following table:

-byte *b* [of] *n*

By default, **gmemfile** outputs all data bytes to the memory image. This option looks at data in the input file in chunks the size of the bus width (*n*) and writes only the specified data bytes (*b*) within each data chunk in the input data to the output file. It can be used to separate odd or even data bytes or words for burning PROMs when the size of the PROMs differs from the size of the bus. It is also used to reverse the order of the data bytes for a shared bus environment.

The width, *n*, represents the width of the data bus. The maximum width allowed is 32 that represents a 256 bit bus. The bytes, *b*, within the specified bus width *n* are numbered 1 through *n* where 1 is the first byte and *n* is the last byte.

The bytes, *b*, can be specified in one of the following ways:

- as a single byte (for example, `-byte 1 of 4`). In this example, for each 4 byte chunk of input data, only byte 1 is written to the output file.
- as a range of bytes (for example, `-byte 1-4 of 8`). In this example, for each 8 byte chunk of data in the input file, bytes 1, 2, 3 and 4 are written to the output file.
- as a list of bytes (for example, `-byte 4,3,2,1 of 4`). In this example, for each 4 byte chunk of data, the bytes are reversed and written to the output file.
- as a combination of a list and a range (for example, `-byte 1-4,6,9-12 of 16`).

-end *address*

Specifies the highest address of data to be placed in the memory image. If *address* is higher than the highest address of the executable, the bytes between them are padded with zeros unless **-fillx** is passed. You can specify a section name instead of an address, and the highest address of that section is used.

-fillx start end value

Fills holes before, after, or between sections of the executable in the specified range *start* to *end* with the specified *value*. The byte order of the input file (big or little endian) is applied to the fill value.

-fill1 fills holes with a single byte of data.

-fill2 fills holes with two bytes of data, aligning the fill bytes on a 2-byte boundary.

-fill4 fills holes with four bytes of data, aligning the fill bytes on a 4-byte boundary.

If the *start* and/or *end* parameters are outside the address range of the data in the executable, the fill value is applied to the gaps at the beginning and/or end of the memory image. Section names can be specified instead of addresses for the *start* and *end* parameters. The start address of a section is used for the *start* parameter. The end address of a section is used for the *end* parameter.

When multiple **-fill** options cover the same area, **-fill4**, **-fill2**, **-fill1** options are processed in that order. The **-start** and **-end** options take precedence over the **-fill** options and can be used to control the bounds of the memory image.

-help

Displays a summary of the command line syntax and options for using **gmemfile**.

-hole size

Overrides the default checking for holes between sections.

By default, a warning message is displayed if there is a hole larger than 4096 bytes; no error is generated for any size hole.

This option causes an error to be emitted if there is a hole larger than *size* bytes.

-import start filename

Copies the binary data from the specified *filename* to the specified *start* address in the memory image. Use the **-import** option to bring in input from a previous run of **gmemfile** or from some other utility that produces a memory image.

If you specify a section name for the *start* parameter, the binary data is copied to the next byte after the last address in the specified section.

-map filename

Creates a detailed map file. The map file contains the command input, a list of all sections from the input executable file, a list of **-import** data, a memory map of the output memory file, and a summary of the memory image.

-o filename

Specifies the name of the memory image file to be output. The default is the name of the executable with the extension, if any, replaced by *.bin*. If there is no input file, the default name of the output file is **a.bin**.

-size value

Sets the size of the memory image to *value* bytes. This is an alternative to using the **-end** option. An error is displayed if both **-size** and **-end** appear in the command input.

By default, all data from the executable is written to the memory image.

-skip sname

Excludes data in the specified section *sname* of the executable from the memory image.

-start address

Specifies the *address* in the executable at which to begin copying data to the memory image. A section name can be specified instead of an address, and then the start address is the first address in the section. If the start address is lower than the lowest address in the executable, the bytes between the start address and the executable are padded with zeros or with a fill value if one was specified with **-fill1**, **-fill2**, or **-fill4**.

-w

Does not display warnings.

Data Splitting

In some hardware implementations, the width of the data bus differs from the width of the PROMs. Depending on how the bus and PROMs are connected, it might be necessary to store low bytes, or words, in one PROM and high bytes, or words, in another PROM. The **gmemfile** utility supports such *data splitting* through the **-byte** option (see Example 19 on page 528 and Example 20 on page 529).

The following examples use for illustration a program file **prog1** that contains the following sections:

Name	Start Address	End Address	Size
.text	0x1000	0x10ff	0x100
.rodata	0x1100	0x12ff	0x200
.rdata	0x1300	0x15ff	0x300
.data	0x2000	0x22ff	0x300
.bss	0x2300	0x24ff	0x200

The `.text` section contains all of the code in the program that is executed from ROM. The `.rodata` section contains read-only data that is accessed in ROM. The `.rdata` section contains initialized data that is stored in ROM and copied to `.data` in RAM at program startup. The program references initialized data at the addresses in section `.data` in RAM. The `.bss` section contains uninitialized data. It is set to zero at program startup.

Example 14. Using gmemfile

```
gmemfile prog1.elf
```

In this example, because no options have been specified, the output file is called **prog1.bin**, the first byte in the memory image is the first byte of section `.text`, and its last byte is the last byte of `.rdata` (which is the last section that contains initialized data).

Example 15. Using the -start and -o Options

```
gmemfile -start 0 prog1.elf -o prog1.mem
```

In this example, the option `-start 0` sets the starting address at which **gmemfile** begins reading the executable to 0. Because the first section of the executable is `.text`, which begins at `0x1000`, the first `0x1000` bytes of the memory image are set to zero. The utility continues reading and outputting the data from sections `.text`, `.rodata`, and `.rdata`. The last byte in the memory image is the last byte of section `.rdata`, which is at file offset `0x15ff` in the memory image and at address `0x15ff` in the executable.

The option `-o prog1.mem` sets the name of the memory image to **prog1.mem**.

Example 16. Using the -fill2 Option

```
gmemfile prog1.elf -fill2 0 .text 0x3e7f -o prog1.mem
```

In this example, the `-fill2` option specifies that any holes between 0 and the end of section `.text` be filled with the value `0x3e7f`. The initialized data in `.text` is not overwritten by `-fill2`. The memory image produced in this example has the same start (0) and end (`0x15ff`) addresses as that in the previous example (the `-start 0` option is not needed, since `-fill2` starts at zero).

Example 17. Using the -start and -end Options

```
gmemfile prog1.elf -start 0 -end 0x1fff -o prog1.mem
```

In this example, the PROM is 0x2000 bytes wide and the PROM burner requires that all bytes in the PROM be filled. The **prog1** executable has a hole in front of its first section (from 0x0 to 0xffff) and after the end of its initialized data (from 0x1600 to 0x1fff).

The `-start` option sets the start address of the memory image to zero and `-end` sets its end address to 0x1fff. Because no `-fill` options have been used, the holes 0x0 to 0xffff and 0x1600 to 0x1fff are filled with zeros.

Example 18. Using the -start and -size Options

```
gmemfile prog1.elf -start 0 -size 0x2000 -o prog1.mem
```

In this example, the memory image produced is identical to that of the previous example. Instead of specifying an end address with `-end`, a size is specified with `-size 0x2000`.

Example 19. Separating Low and High Bytes of Data into Separate PROMs

```
gmemfile -byte 1 of 2 prog1.elf -start .text -size 0x100 -o prog1.low.mem  
gmemfile -byte 2 of 2 prog1.elf -start .text -size 0x100 -o prog1.high.mem
```

In this example it is necessary to separate low and high bytes of data into separate PROMs, each of which is 0x100 bytes wide. The **gmemfile** utility is run twice, using the `-byte` option to select alternate bytes for each of the output files.

In the first command line, the option `-byte 1 of 2` outputs the first byte of data in each two byte interval. The `-start` option sets **gmemfile** to reading the executable at 0x1000 (the first byte of `.text`). The `-size` option sets the size of the memory image to 0x100 bytes.

The first byte of data in the memory image is the first byte of data in section `.text`. The second byte of data in the memory image is the third byte of data in section `.text`. The last byte of data in the memory image at file offset 0xff is from the `.rodata` section in the executable at address 0x11fe (0x200 bytes above 0x1000).

If the bytes in the `.text` section of the executable consisted of:

```
0102030405060708090a0b0c0d0e0f10....
```

The following bytes would be output to the first memory image file:

```
01030507090b0d0f....
```

The second command line takes the second byte in each 2-byte interval starting at the address of `.text` (0x1000). The first byte of data in the memory image is the second byte of data in section `.text`. The second byte of data in the memory image is the fourth byte of data in section `.text`. The last byte of data in the memory image is the byte in section `.rodata` at address 0x11ff.

If the bytes in the `.text` section of the executable consisted of:

```
0102030405060708090a0b0c0d0e0f10....
```

The following bytes would be in the second memory image file:

```
020406080a0c0e10....
```

Example 20. Separating the Low Half and High Half of Each 4-byte Interval

```
gmemfile -byte 1,2 of 4 prog1.elf -start .text -size 0x100 -o prog1.low.mem  
gmemfile -byte 3,4 of 4 prog1.elf -start .text -size 0x100 -o prog1.high.mem
```

This example is similar to the previous example, except that the low half and high half of each 4-byte interval is separated. The first command takes the low 2-byte half of each 4-byte interval of data in the executable starting at the address of `.text` (x1000) until there are 0x100 bytes of data in the memory image.

If the bytes in the `.text` section of the executable consisted of:

```
0102030405060708090a0b0c0d0e0f10....
```

The following bytes would be in the first memory image file:

```
01020506090a0d0e....
```

The second command takes the high 2-byte half of each 4-byte interval of data in the executable starting at the address of `.text` (x1000) until there are 0x100 bytes of data in the memory image.

If the bytes in the `.text` section of the executable consisted of:

```
0102030405060708090a0b0c0d0e0f10....
```

The following bytes would be in the second memory image file:

```
030407080b0c0f10....
```

Example 21. Changing Data in the `.rdata` Section

This example shows you how to change the endianness of the data in the `.rdata` section of **prog1.elf**.

```
gmemfile prog1.elf -start .rdata -end .rdata -byte 4,3,2,1 of 4 -o reverse.bin
gmemfile prog1.elf -skip .rdata -import .rodata reverse.bin -o prog1.bin
```

The first command loads just the `.rdata` section of **prog1.elf** into memory. The `-byte 4,3,2,1 of 4` option reverses the byte order of the data. Then, **gmemfile** writes the result to the memory image file **reverse.bin**.

The second command loads **prog1.elf** into memory, but the `-skip .rdata` option discards any data in the `.rdata` section. The `-import .rodata reverse.bin` option loads **reverse.bin** after the end of the `.rodata` section (the beginning of the `.rdata` section). Finally **gmemfile** writes the result to **prog1.bin**.



Note Because the `-import` option instructs **gmemfile** to place the data in **reverse.bin** immediately after the `.rodata` section, this example works only if the end of the `.rodata` section is one byte before the beginning of the `.rdata` section.

The *gnm* Utility Program

The **gnm** utility program performs a similar function to the Linux/Solaris **nm** program. It displays the symbol table of an object file, object library file, or executable file created by Green Hills development tools.

The syntax for this utility is as follows:

gnm [*options*] *filenames*...

where *filenames* is one or more whitespace-delimited input files, and *options* are listed in the general options table and the appropriate object format options table:

General Options

The general **gnm** options are as follows:

-help	Displays general options and ELF-specific options.
-helpbsd	Displays BSD-specific options.
-helpcoff	Displays COFF-specific options.
-helpelf	Displays ELF-specific options.
-output <i>file</i>	Writes the output from gnm to <i>file</i> .
-V	Prints the gnm version.

ELF File Options

The **gnm** options for ELF format files are as follows:

-a	Prints special symbols that are normally suppressed.
-D	Prints symbols from the dynamic symbol table.
-defined_only	Prints only defined symbols.
-dg	Prints defined symbols and global symbols. It does not print debug symbols.
-g	Prints only global symbols.
-h	Does not print headers.
-l	Prints an asterisk (*) after symbol type for WEAK symbols. This is for -p mode only.
-n	Sorts output by symbol name.
-no_debug	Does not print debug symbols such as <code>..bof</code> , <code>..eof</code> , and <code>..lin</code>
-no_dotdot	Does not print any symbols beginning with <code>..</code> .
-no_line	Does not print line debug symbols. For example, <code>..lin</code> .
-o	Prints value and size of symbols in octal.
-p	Uses 3-column output format.
-prefixed_msgs -no_prefixed_msgs	Enables or disables the printing of <code>WARNING</code> and <code>ERROR</code> before messages.
-r	Prefixes filename to each line of output.

-R
Prefixes archive name and filename to each line of output.
-S
In 3-column output mode, -S prints section names instead of one-letter codes.
-u
Prints undefined symbols only.
-v
Sorts output by symbol value.
-X
Does not print local compiler labels (<i>.Ln</i>).
-x
Prints value and size of symbols in hexadecimal.

BSD File Options

The **gnm** options for BSD format files are as follows:

-a
Prints all symbols, including debugging information. Not available with the -s option.
-g
Prints only external symbols.
-n
Sorts symbols by value.
-o
Prefixes lines with object or archive element name.
-p
Does not sort symbols.
-r
Sorts in reverse order.
-s
Sorts external symbols by size.
-u
Prints only undefined symbols.

COFF File Options

-a	Prints special symbols that are normally suppressed.
-A	Uses System V output format.
-E	Treats input files as Alpha ECOFF format.
-g	Prints only global symbols.
-h	Does not print headers.
-p	Uses 3-column output format.
-r	Prefixes filename to each line of output.
-s	Prints section names.
-t	Truncates standard call suffixes (Win32).
-T	Truncates names to 20 characters.
-u	Prints only undefined symbols.

Output Formats

By default, **gnm** produces output similar to the following excerpt (indices 3 through 11 are omitted for brevity):

[Index]	Value	Size	Type	Bind	Other	Shndx	Name
[1]		0	0 FILE	LOCL	0	ABS	hi.c
[2]		0	0 SECT	LOCL	0	.text	.text
[12]		0	52 FUNC	GLOB	0	.text	main
[13]		0	0 NOTY	GLOB	0	UNDEF	printf

Column	Meaning
Index	Position of symbol in the symbol table.
Value	The value or address of the symbol.
Size	Size of the symbol (for example, a 4-byte integer symbol has a size of 4).
Type	One of the following: • NOTY: Symbol without type. • FILE: Filename symbol. • SECT: Section name symbol. • OBJT: Data symbol. • FUNC: Code symbol.
Bind	One of the following: • LOCL: Local symbol (for example, C/C++ static). • GLOB: Global symbol. • WEAK: Weak global symbol (if undefined, the symbol's value resolves to zero).
Other	Reserved field, generally zero.
Shndx	The name of the section where the symbol is defined. For example: • .text: Code defined in the .text section. • .data: Data defined in the .data section. • ABS: No section (for example, a filename symbol). • COMMON: Common variable whose section is not yet determined.
Name	Symbol name.

The **-p** option enables an alternative, easily parsable, 3-column format, which provides backward compatibility with tools that can read only this format:

```
0000000000 F hi.c
0000000000 S .text
0000000000 T main
0000000000 U printf
```

The first column is the value or address of the symbol. The second column is its type (as shown in the following table), and the third is the symbol name. Note that this format cannot fully describe all sections and storage classes.

Type	Meaning
A	External absolute
a	Local absolute
B	External zeroed data
b	Local zeroed data
C	Common variable (same as B except not yet assigned to a section)
D	External initialized data
d	Local initialized data
F	Filename
I	Local thread local-storage
L	External thread-local storage
N	Informational symbol
S	Section name
T	External text
t	Local text
U	External undefined

The gpjmodify Utility Program

The **gpjmodify** utility program allows you to automate editing of project **.gpj** files. General editing should be performed through the Project Manager (see Chapter 2, “The MULTI Project Manager”), or via a text editor.



Note You can edit only one file at a time using **gpjmodify**.

Editing Existing .gpj Files

The syntax for editing existing **.gpj** files is as follows:

```
gpjmodify file.gpj [[sub.gpj...] | [@path-to-sub]] [options] action
```

where:

- **file.gpj** is a Top Project file. This is either the file to be edited or the ultimate parent of **sub.gpj**.
- The optional **sub.gpj...** is an immediate child subproject of **file.gpj** or a chain of subprojects (the last of which will be edited). Alternately, you can specify **@path-to-sub.gpj**, an absolute path to a subproject at any point in the project tree.
- **options** are listed in the following table:

-f	Forces the edit, even if errors would result.
-help	Displays help information.
-multidir <i>directory</i>	Specifies an alternative MULTI installation <i>directory</i> .
-o <i>output.gpj</i>	Writes the edited file to output.gpj . By default, edits are written to the existing file.
-quiet	Suppresses warning messages.

- **action** listed in the following table:

-add_after <i>child</i> [-filetype <i>type</i>] <i>files</i> [-sourcedir <i>dir_paths</i>]
Inserts <i>files</i> in the project's list of children, directly after <i>child</i> .
-add_before <i>child</i> [-filetype <i>type</i>] <i>files</i> [-sourcedir <i>dir_paths</i>]
Inserts <i>files</i> in the project's list of children, directly before <i>child</i> .
-add_to_back [-filetype <i>type</i>] <i>files</i> [-sourcedir <i>dir_paths</i>]
Inserts <i>files</i> at the end of the project's list of children.
-add_to_front [-filetype <i>type</i>] <i>files</i> [-sourcedir <i>dir_paths</i>]
Inserts <i>files</i> at the beginning of the project's list of children.
-change_macro <i>name=value</i>
Changes the value of the existing macro <i>name</i> to <i>value</i> .
-remove <i>files</i>
Deletes the specified <i>files</i> from the project's list of children.
-replace_with_existing_child [-filetype <i>type</i>] <i>new_child</i>
Replaces the references to the file <i>existing_child</i> with references to <i>new_child</i> .
-set options
Inserts a list of build <i>options</i> .
-set_target <i>target_file</i>
Specifies the project's target file. This is only saved if you are modifying a Top Project file.
-unset options
Deletes a list of build <i>options</i> .

Pass the **-filetype *type*** parameter only if the files you are adding have non-standard extensions that the Project Manager is unable to recognize. For a list of accepted values, see “File Types” on page 39. If *type* contains a space, you must enclose it in quotes.

The **-sourcedir *dir_paths*** option specifies paths in which to search for the specified *files*. **gpjmodify** always uses the first file that it finds.

For any of the actions that require a list of *files* or *options*, each item in the list must be separated with a space.

Creating a New .gpj File

The syntax for creating a new **.gpj** file is as follows:

gpjmodify *file.gpj* -create *type* -target *target_file* [*options*]

where *file.gpj* is the name of the file to be created, *type* is the type of project file to create (usually Project, Subproject, Program, or Library), and *target_file* specifies the project's target. Valid *options* are listed in the following table:

-multidir <i>directory</i>
Specifies an alternate MULTI installation <i>directory</i> .
-quiet
Suppresses warning messages.
-top
Specifies that the new file will be a Top Project project file.

Generating a Makefile

Normally, if you need to build your project from the command line, you should use the **gbuild** utility program, which provides a command line interface to build MULTI projects (see “The gbuild Utility Program” on page 499). However, if you need to integrate Green Hills tools with an existing makefile structure, the **gpjmodify** utility can generate a rudimentary makefile from a project (**.gpj**) file. Use this makefile as a guide when writing makefiles for your project.

To generate a makefile with **gpjmodify**, enter the following command:

gpjmodify *file.gpj* -generate_makefile [*options*]

where *file.gpj* is the **.gpj** file to be converted, and valid *options* are listed in the following table:

-multidir <i>directory</i>
Specifies an alternate MULTI installation <i>directory</i> .
-quiet
Suppresses warning messages.



Tip If you also need to know the commands **gbuild** executes when building your project, enter the following command:

gbuild -info -commands *file.gpj*

The grun Utility Program

The **grun** utility program downloads and remotely executes a program using a debug server to control the execution environment. While **grun** is running, its standard input and output are copied to and from the executing program, redirecting the program's I/O to your terminal.

The syntax for this utility is as follows (note the use of the double dash “- -”):

grun [*options*] [*dbserv_cmd*] -- *program_cmd*

where:

- *dbserv_cmd* is the name of a debug server and any arguments
- *program_cmd* is the name of the executable and any arguments
- *options* are listed in the following table:

-commands=<i>file</i>
Sends the commands specified in <i>file</i> to the debug server before downloading the program.
-data <i>address</i>
Sets the starting <i>address</i> of the program's data.
-detach
Instructs grun to terminate immediately after downloading the program to the target system. The grun program usually communicates with the debug server before exiting, which can halt an executing target program. This option can be used with various target monitors or boot ROM's when the program being downloaded takes control of the target system and terminates the target monitor. If you do not specify the -detach option, grun waits either for the program to complete, or for up to 3 minutes with no host I/O requests from the program. If you interrupt grun , or if the program exceeds the 3 minute time out, grun terminates the program and exits.
-download
Causes the program to be downloaded, but does not start it running.
-log[=<i>file</i>]
Outputs a log of the Debugger and debug server communications to <code>stdout</code> , or to <i>file</i> , if specified.
-pro
Same as the -profile option, but also translates the profiling data.

-profile

Executes the target program with profiling enabled.

-stack *address*

Sets the initial value for the program's stack pointer to *address*.

-suppress_exit_message

Instructs **grun** to not print the program's exit value.

-text *address*

Sets the starting *address* of the program's text (code).

-timeout=*seconds*

Sets the number of seconds to wait before assuming the target has hung. The default is 180 seconds.



Note **grun** is only supported by stop-mode debug servers such as **mpserv**. **grun** is not supported with **rtserv** (INTEGRITY 5), **rtserv2** (INTEGRITY 10), or any native debug server.

The gsize Utility Program

The **gsize** utility program performs a similar function to the Linux/Solaris **size** program. It analyzes object files, object library files, or executable files and, for each file, displays the size of each section in bytes. If more than one file is named, or if an object library is named, **gsize** prints the name of the file with the section name and totals for each section.

The syntax for this utility is as follows:

gsize [*options*] *filename*

where *filename* is the name of the input BSD or ELF object file, object file library, or executable file.

The *options* for **gsize** are as follows.

-all Causes all sections to be displayed, including empty sections and non-allocated sections.
-commons Displays a block of information describing each <code>COMMON</code> symbol in each output file (see “Text and Data Placement” on page 127). This block is suppressed if you pass the -table , -nodetails , or -nobss options. This option implies -count_commons .
-count_commons Includes the size of common variables in the total size of <code>.bss</code> , the size of zero-common variables in the total size of <code>.PPC.EMB.sbss0</code> , and the size of small-common variables in the total size of <code>.sbss</code> . This option has no effect if you pass the -bss option.
-details — [default] -nodetails Displays or suppresses detailed information about each section in each file.
-help Displays information about all options.
-nobss Suppresses <code>bss</code> sections. This option makes -commons and -count_commons have no effect.
-nodata Suppresses <code>data</code> sections.

-notext

Suppresses `text` sections.

-ram

Displays a sum total of RAM size for the input files. File segments are considered to be RAM only if they are writable and not executable.

A section can be both a RAM and a ROM section. For example, the `.data` section in an ELF object file is writable, not executable, and contains initialized data.

This option must be passed alone (or in conjunction with **-rom**). For example:

```
> gsize -ram kernel
    RAM_Size:      3904
```

-rom

Displays a sum total of ROM size for the input files. File segments are considered to be ROM only if they contain initialized data or program code.

A section can be both a RAM and a ROM section.

This option must be passed alone (or in conjunction with **-ram**). For example:

```
> gsize -rom kernel
    ROM_Size:      37818
```

or:

```
> gsize -rom -ram kernel
    RAM_Size:      3904
    ROM_Size:      37818
```

-table

Displays output in table format. This option implies **-details** and **-nototals**.

-text

Only displays text sections, suppressing `data` and `bss` sections. This is the same as **-nodata** and **-nobss**.

-totals — [default]**-nototals**

Displays or suppresses the summary information. If you pass **-totals**, summary information is output only if **gsize** is processing multiple files or common symbols.

-zero

Displays zero-length sections.

The gsrec Utility Program

The **gsrec** utility program converts an executable file into Motorola S-Record, Intel hexadecimal, or Tektronix hexadecimal format. These file formats contain ASCII representations of binary data. Many simulators, In-Circuit Emulators (ICEs), PROM programmers, and debuggers use one of them as a program download format. Use the **-hex** and **-srec** driver options to automatically invoke **gsrec** and generate a Intel hexadecimal or S-Record file. For more information about these options, see “**Generate Additional Output**” on page 244.

The syntax for this utility is as follows:

gsrec [*options*] *filename*

where *filename* is the name of the input executable file to be converted to one of the output formats. Motorola S-Records is the default output format.

The **gsrec** *options* are:

-auto
Determines byte order from the file header. This is the default.
-B
-L
Specifies a big or little endian input file.
-bytes <i>n</i>
Sets the maximum count of unpaired data bytes/records (minimum 4, maximum 28). The default is 28.
-e <i>addr</i>
Sets entry point in the termination record to given address.
-end <i>address</i>
Sets end <i>address</i> in object file.
-eol <i>char</i>
Specifies the character used to terminate each S-Record, where <i>char</i> is one of: • cr: a <code>\r</code> character. • crlf: a <code>\r\n</code> combination. This is the default in Windows. • lf: a <code>\n</code> character. This is the default in Linux/Solaris.

-fillx n1 n2 v

Fills memory from address *n1* to address *n2* with the *x*-byte value *v* (where *x* is 1, 2, or 4).

-help

Displays information about all options.

-hex86**-hex386****-tekhex**

Specifies output of Intel HEX86, HEX386, or extended Tektronix hexadecimal format.

The Intel HEX86 hexadecimal format can handle memory addresses only in the range of 0x00000000 to 0x0010ffef. When this address range is exceeded, **gsrec** displays the message: `an address is too big`.

The Intel HEX386 format supports the full 32-bit address space. A linear address record containing the upper 16-bits of the address was added to the HEX86 format to form the HEX386 format.

The Tektronix extended tekhex format supports the full 32-bit address space. The **-tekhex** option is deprecated and may be removed in future versions of **MULTI**.

-interval n[:m]

Places only those data bytes from the input file that occur within the specified interval in the output file. Values for *n* are 1 through 8. The default is 1, indicating that every byte will be output. The *m* value specifies the number of output bytes for each interval. For example, `4 : 2` outputs 2 consecutive bytes out of every 4.

This option requires that you set the **-start** and **-romaddr** options.

-noS5

Suppresses production of an *S5* block count record.

-noSLA

Do not output the Start Linear Address Record in HEX386 mode. Instead, the start address is placed in the end record.

-o filename

Specifies the output filename. By default, output is sent to standard output.

-padx n1 n2 v

Pads *x*-byte holes between hexadecimal data records from address *n1* to *n2* with value *v*. *x* may be 1, 2, or 4.

-prefixed_msgs
-no_prefixed_msgs Enables or disables the printing of WARNING and ERROR before messages.
-raw Treats the input file as raw data to be placed at the address specified by -romaddr .
-romaddr <i>addr</i> Sets start address in ROM to place data. The default is the same address as in the input file.
-Stype Specifies the type of S-Record to be produced, where <i>type</i> is one of the following: • 1: S1 data records (16-bit addresses) • 2: S2 data records (24-bit addresses) • 3 (default): S3 data records (32-bit addresses) • 5: an S5 record with a count of data records (32-bit count) • 5old: an old format S5 record (16-bit count) • 7: an S7 end record (32-bit entry point) • 8: an S8 end record (24-bit entry point) • 9: an S9 end record (16-bit entry point) The default is to produce S3 data records, an S5 record, and an S7 end record.
-skip <i>s</i> Does not output data for section <i>s</i> .
-sort Sorts output records in ascending order.
-start <i>address</i> Specifies starting <i>address</i> in object file for output.
-w Does not issue a warning when the input is a relocatable object file.
-warn_overlap Issues warnings for input sections that overlap.

S-Record Output Format

An S-Record file contains ASCII text that can be displayed or edited. There are eight types of S-Records, numbered S0 to S9, including the following:

- An S0 header record that identifies the program.
- Data records that represent the binary data in the program.
- An S5 data count record that contains the count of data records in the S-Record file.
- A termination record that contains the address to begin program execution.

Not all S-Record reader programs behave the same way. Some programs require certain types of data records. Some target environments do not accept S5 data count records, and some require certain types of termination records. The **gsrec** utility provides options that handle many of these cases.

Every S-Record begins with the letter S followed by a digit (0 through 9) representing the format of the line. The next two characters are a hexadecimal representation of the number of bytes in the remainder of the line, which always consists of the fields referred to as address, data, and checksum. The address field may contain 2, 3, or 4 bytes depending on the format. The length of the data field varies. The checksum field always contains 1 byte.

For example, the following S3 record has a length of 33 (0x21) bytes.

```
S3 21 00000000 00E280FF04001FE80848074006384036000006360000B20580FF0000 B5
```

The 4-byte address is 00000000, the data field contains 28 bytes, and the checksum is one byte.

Data and Termination Records

A data record contains the address where data is loaded, followed by the data itself. The three types of data records, S1, S2, and S3, differ in load address size, as indicated in the previous table.

By default, an S5 data count record is emitted. If your S-Record loader does not handle S5 records, use the option **-noS5** to avoid outputting an S5 data count record.

By default, the S5 data record contains a 4-byte address. Some old S-Record readers expect an S5 record to contain a 2-byte address. In this case, use the **-S5old** option.

A termination record contains the entry point (the address where program execution begins). There are three types of termination records, S7, S8, and S9, as indicated in the previous table.

Some S-Record readers accept data records up to a specified length. Use the option **-bytes** *size* to set the maximum data record length.

If you forget to specify the entry point address when linking, you can use **-e** *address* to set the entry point to the given *address*.

Data Splitting

In some hardware implementations, the width of the bus differs from the width of the PROMs. Depending on how the bus and PROM's are connected, it might be necessary to store even bytes in one PROM and odd bytes in another PROM. *Data splitting* is a technique of dividing the data into even and odd bytes, or in other ways.

gsrec can perform data splitting. The **-start** option specifies the starting address of the data in the object input file to output to the S-Record output file. The **-end** option specifies the last address of data in the object input file to output. The **-interval** option specifies the distance between bytes in the input file to output. A value of 2 for **-interval** outputs every other byte. Sometimes it is necessary to relocate the data to address zero in the S-Record output file for programming PROMs. The **-romaddr** option specifies the start address of the data bytes in the S-Record output file. The last two succeeding examples separate even and odd bytes into two different S-Record files.

The following examples use a program file **prog1** that contains the following sections:

```
1: .text,    address: 0x1000, size: 0x100
2: .data,    address: 0x2000, size: 0x200
3: .bss,     address: 0x3000, size: 0x200 (No Load Section)
4: .data2,   address: 0x4000, size: 0x200
```

Example 22. Using gsrec

```
gsrec prog1
```

The **gsrec** program reads the data in the input file **prog1** and writes the S-Records to the standard output. Because the contents of section **.bss** are not loaded, no S-Records are output for its data.

Example 23. Using the *-Stype* Options

```
gsrec -S1 -S9 prog1 -o prog1.run
```

The **gsrec** program reads the data in the input file **prog1** and writes the S-Records to the specified output file **prog1.run**. Data are output as S1 records that have a 16-bit address space. The start address is output as an S9 record that has a 16-bit address space.

Example 24. Using the *-start*, *-end*, and *-o* Options

```
gsrec -start 0x1000 -end 0x1080 prog1 -o prog1.run
```

The **gsrec** program reads the data in the input file **prog1** starting at address 0x1000 and ending at address 0x1080 and writes the resulting S-Records to the output file **prog1.run**.

Example 25. Using the *-romaddr* Option

```
gsrec -start 0x1000 -end 0x1080 -romaddr 0 prog1 -o prog1.run
```

The **gsrec** program reads the data in the input file **prog1** starting at address 0x1000 and ending at address 0x1080, relocates the data to start at address zero, and writes the resulting S-Records to the output file **prog1.run**.

Example 26. Using the *-interval* Option for Even Data Bytes

```
gsrec -start 0x1000 -end 0x1080 -romaddr 0 -interval 2 prog1
```

The **gsrec** program reads the data bytes at even addresses in the input file **prog1** starting at address 0x1000 and ending at address 0x1080, relocates the data to start at address zero, and writes the resulting S-Records to the standard output.

Example 27. Using the *-interval* Option for Odd Data Bytes

```
gsrec -start 0x1001 -end 0x107F -romaddr 0 -interval 2 prog1
```

The **gsrec** program reads the data bytes at odd addresses in the input file **prog1** starting at address 0x1001 and ending at address 0x107F, relocates the data to start at address zero, and writes the resulting S-Records to the standard output.

Example 28. Using the -interval Option for 2 Out of Every 4 Data Bytes

```
gsrec -start 0x1002 -end 0x107F -romaddr 0 -interval 4:2 prog1
```

The **gsrec** program reads 2 out of every 4 data bytes in the input file **prog1** starting at address 0x1002 and ending at address 0x107F, relocates the data to start at address zero, and writes the resulting S-Records to the standard output.

The gstack Utility Program

The **gstack** utility program analyzes a program to report the maximum stack size each task might need during execution and the call chain that produces this maximum stack size.

To produce the information required for **gstack**, all files comprising the program must be compiled with **-G**. All functions in assembly files must be annotated with the `.fsize` and `.scall` directives or have their sizes and calls declared with the **-f** and **-a** options to **gstack**.

The syntax for this utility is as follows:

gstack *options program*

where *program* is the name of an input executable program to examine.

The **gstack** options are:

-a
Adds functions or connections to the call graph. -a fun1:fun2, fun3 adds fun1, fun2, fun3 to the call graph, with fun2 and fun3 being called by fun1 .
-c func
Displays all callers of <i>func</i> .
-d callee[:caller,...]
If you specify one function, this option deletes all calls to that function. For example, -d foo deletes all calls to <code>foo()</code> .
If you also specify a list of callers, this option deletes all calls from those callers to the specified callee. For example, -d foo:bar,baz deletes all calls from <code>bar()</code> and <code>baz()</code> to <code>foo()</code> .
-e
Specifies an entry point to start from
-f func=size
Specifies function stack frame <i>size</i> .
-g
Displays the call graph.
-help
Displays information about all options.

-j
Displays all functions and frame sizes.
-missing_fsize
Displays all functions that are missing static call information.
-missing_scall
Displays all functions that are missing frame size information.
-recursive
Displays all functions that potentially call themselves recursively (directly or indirectly).
-s func
Displays the maximum stack size for the program, starting with <i>func</i> .
-u
Displays all functions that have no callers.

Caveats

- **gstack** cannot accurately calculate the maximum stack size when there are potential recursive (direct or indirect) calls in the chain, because it cannot predict how many times the recursion will occur. **gstack** prints a warning message if it detects a possible recursion and assumes those calls will not occur in its calculations.
- **gstack** prints a warning message if it detects a call to a function for which there is no frame size or static call information. This information is necessary in order for **gstack** to provide accurate results. This information may be missing because the function was compiled without the **-G** option, or because it was written in assembly, but lacks the proper annotation.
- **gstack** does not understand function calls through function pointers, including C++ virtual function calls. No warning is printed. To denote functions that may be called through a function pointer, use the `#pragma ghs static_call` directive in C and C++, or the `.scall` directive in assembly. Alternatively, use **gstack -a**.
- **gstack** cannot differentiate between static functions of the same name from different files.
- **gstack** analysis cannot detect changes to your program made by **elxr** as a result of using the **-wrap** option. If you passed the **-wrap** option to **elxr**

when linking your program, pass the **-a** and **-d** options to **gstack** to reflect these changes.

- Use of **gstack** is not supported with programs that were built by third-party tools, even if the associated debugging information has been translated into the MULTI **.dbo** format using **dwarf2dbo** or **stabs2dbo**.

The **gstrip** Utility Program

The **gstrip** utility program can remove line number, symbol table, and debugging information from an executable file to reduce its file size on disk.

The **gstrip** utility processes executable files created by the Green Hills Software linker as well as those created by some native linkers.

The syntax for this utility is as follows:

gstrip [*options*] *filename*

where *filename* is the name of an executable file.

The *options* for **gstrip** are as follows:

-help
Displays information about all the options.
-l
Removes line number information only, without stripping the symbol table or debugging information.
-prefixed_msgs -no_prefixed_msgs
Enables or disables the insertion of the words <code>WARNING</code> and <code>ERROR</code> before every warning or error message.
-V
Displays the version number of gstrip to standard error output.
-x
Does not remove the symbol table; debugging and line number information might be removed.

The gversion Utility Program

The **gversion** utility program extracts and prints date and time information from an executable file provided by Green Hills. If no options or slots are specified, then **gversion** outputs the executable's revision and release dates from slots 0 and 6.

The syntax for this utility is as follows:

gversion [*options*] [*slot*] [*file1*] [*file2* ...]

where:

- *options*: Listed in the table below.
- *slot*: A single digit number, specifying the slot to print (see Example 29 on page 556). **gversion** displays the file modification date when no valid time stamp corresponding to the slot requested can be found.
- *file1*, *file2*: The executable files for which you want to print date and time information.

-all
Displays all non-zero dates, marked [0] through [9]. Each date is preceded by a digit in square brackets "[]" called a time stamp. The revision date is preceded by [0] and the release date is preceded by [6]. See Example 30 on page 556.
-date (default)
Displays the value as a date string.
-help
Displays information about all options.
-mtime
-nomtime (default)
Enables or disables display of the file's modification date. The -mtime option must be used with -all . See Example 31 on page 556.
-quiet
Suppresses errors for files that are not stamped.
-value
Displays all stamp values without converting to date.

Example 29. Using gversion

In this example, suppose the executable **gversion** has only time slot 0 set:

```
> ./gversion gversion
gversion: Green Hills Software, MULTI v5.0.6
gversion:                               Revision Date Mon Nov  9 10:48:25 PST 2009
gversion:                               Release Date  Mon Dec 14 17:08:14 PST 2009
```

If asked to print the value of slot 1, which is not stamped, **gversion** displays an error message:

```
> ./gversion 1 gversion
gversion: Green Hills Software, MULTI v5.0.6
gversion has not been time stamped
gversion:                               [m] Mon Nov  9 10:48:25 2009
```

Example 30. Using -all

When using **-all**, **gversion** displays all non-empty time stamps.

```
> ./gversion -all gversion
gversion: Green Hills Software, MULTI v5.0.6
gversion:                               [0] Mon Nov  9 10:48:25 PST 2009
gversion:                               [6] Mon Dec 14 17:08:14 PST 2009
gversion: License Managed by Green Hills License Manager
```

Example 31. Using -mtime

The **-mtime** option must be used with **-all**. It provides the file's modification date.

```
> ./gversion -mtime -all gversion
gversion: Green Hills Software, MULTI v5.0.6
gversion:                               [0] Mon Nov  9 10:48:25 PST 2009
gversion:                               [6] Mon Dec 14 17:08:14 PST 2009
gversion: License Managed by Green Hills License Manager
gversion:                               [m] Mon Nov  9 10:48:25 2009
```

The gwhat Utility Program

The **gwhat** utility program performs a similar function to the Linux/Solaris **what** program. It reports or updates the version information of a file.

The syntax for this utility is as follows:

gwhat [*options*] [*file1*] [*file2* ...]

The options to **gwhat** are as follows:

-all	Prints out all <code>what</code> -strings.
-c <i>what_str</i>	Overwrites the default <code>what</code> -string space reserved for customers with <i>what_str</i> .
-f	Overwrites the <code>what</code> -string, even if it is read-only.
-help	Displays help message.
-m <i>what_str</i>	Overwrites the Green Hills <code>what</code> -string space with <i>what_str</i> .
-sn	Stops after the <i>n</i> th occurrence of the pattern. The default is 1 (first occurrence). There is no space between the -s and <i>n</i> .
-w <i>what_str</i>	Changes the <code>what</code> -string marker from <code>@(#)</code> to <i>what_str</i> .

If neither the **-c** nor **-m** option is specified, then by default **gwhat** outputs the first string in the file. You can set the number of strings to output with the **-all** or **-s** option.

Version Extract Mode

In version extract mode, **gwhat** searches each filename for occurrences of the following 4-byte `what`-string marker:

`@(#)`

It then displays all subsequent characters up to but not including any of the following terminator characters (expressed in C) or end-of-file:

Character	Character name
"	Straight double quotation mark
>	Greater than sign
\\	Backslash
\n	Newline
\0	Null character

For example, if the C program in file **program.c** contains:

```
char sccsid[] = "@(#)Version 3.0";
```

and if **program.c** is compiled to yield **program.o** and linked to yield **a.out**, then the command:

```
gwhat program.c program.o a.out
```

produces:

```
program.c:
    Version 3.0
program.o:
    Version 3.0
a.out:
    Version 3.0
```

Version Update Mode

In version update mode, **gwhat** can write new version information into a file. This feature has several applications. All Green Hills executables are built with two default what-strings. For example:

```
"@(#)Green Hills Software, Version 4.0"
"@(#) [Customer version stamp space]"
```

Under some circumstances, Green Hills may change the Green Hills release what-string, using the **-m** option. For example:

```
gwhat -m 'special release abc' elxr
gwhat -s2 elxr
      Green Hills Software, special release abc
      [Customer version stamp space]
```

This capability is available to Green Hills customers as well, although it is recommended that the Green Hills release what-string be left unchanged.

If you want to stamp a Green Hills executable with a private identification mark, change the [Customer version stamp space] that is provided for this purpose. For example:

```
gwhat -c 'Mary likes this version' elxr
gwhat -s2 elxr
      Green Hills Software, Version 4.0
      Mary likes this version
```

The **-c** option can only be used once on a binary because it looks for the original [Customer version stamp space] identifier.

You might want to build a custom what-string into your own software to mark the executables. For example, suppose your executable, **acmeprog**, contains a C source file that contains:

```
char sccsid[] = "@(#)ACME SOFTWARE          ";
```

The following shows the different outputs of **gwhat** before and after using the **-m** and **-w** options.

```
gwhat acmeprog
      ACME SOFTWARE
gwhat -w "@(#)ACME SOFTWARE " -m "version 12" acmeprog
gwhat acmeprog
      ACME SOFTWARE version 12
```


Language Reference

Part III

Green Hills C

Chapter 14

This Chapter Contains:

- Specifying a C Language Dialect
- ANSI C
- Strict ANSI C
- GNU C
- Strict ISO C99
- ISO C99
- K&R C
- Motor Industry Software Reliability Association (MISRA) Rules
- Japanese Automotive C Extensions
- Type Qualifiers
- Thread-Local Storage for Cafe OS
- Assignment and Comparisons on struct and union Types
- Bitfields
- Enumerated Types
- Functions with Variable Arguments
- The asm Statement
- The Preprocessor
- Extended Characters
- Compiler Limitations
- ANSI C Implementation-Defined Features

This chapter describes the Green Hills implementation of C, including aspects of the language particular to our compilers.

Specifying a C Language Dialect

The Green Hills C Compiler supports six main variants of the C Language.

To specify a C dialect:

Set the **C/C++ Compiler→C Language Dialect** option to one of the following settings:

- **Strict ISO C99 (-C99)** — Accepts the ISO C99 language as defined by *ISO/IEC 9899:1999* and does not support extensions.
- **ISO C99 (-c99)** — Accepts the ISO C99 language with extensions. Since the ISO C99 language itself contains numerous useful features which previously were only available as extensions, this mode is useful for development of entirely new projects (see “ISO C99” on page 579).
- **Strict ANSI C (-ANSI)** — Accepts the ANSI C language as defined by *X3.159–1989* and does not support extensions.
- **ANSI C (-ansi)** — [default] Recommended for all new development, and for the compilation of any existing C code which approximates to ANSI C. For more information, see “ANSI C” on page 565.
- **GNU C (-gcc)** — Recommended for porting existing GNU code to the Green Hills development environment. For more information, see “GNU C” on page 573.
- **K&R C (-k+r)** — Support for K&R mode is deprecated and may be removed in future versions of MULTI. Recommended only for porting existing K&R code to the Green Hills development environment. For more information, see “K&R C” on page 582.

The ANSI C standard can be considered the base of all dialects. Each dialect provides certain extensions and features in addition to the basic ANSI language, except K&R C, which omits some features and allows various non-standard constructs. If your program is written in standard ANSI C, any of the modes except for K&R C should compile your program correctly with few exceptions. All dialects use the same library and header files, which provides the complete ISO C99 library. A few header files and their associated functions are only supported in the ISO C99 dialect. We strongly recommend that you compile

all modules in a particular program using the same dialect, as a failure to do so could result in obscure failures at link time or run time.



Note If you are using the Strict ANSI C, ANSI C, or GNU C dialects, `strtof()` uses the ANSI C semantics, meaning that on underflow it returns zero and sets `errno` to `ERANGE`. `strtod()` and `strtold()` return the smallest non-zero value with the correct sign and set `errno` to `ERANGE`.

In addition to the six primary C dialects, we also provide support for:

- the Motor Industry Software Reliability Association (MISRA) C rules (see page 197)
- the Japanese Automotive C extensions (see page 587)

ANSI C

The ANSI C dialect is the default dialect. We recommend that you use this mode for all new development, as well as for the compilation of any existing C code that comes close to ANSI C.

To use this dialect:



Set the **C/C++ Compiler→C Language Dialect** option to **ANSI C** (`-ansi`).

The compiler accepts the ANSI C mode as defined by *X3.159–1989*, with the addition of support for numerous extensions. These extensions may be useful, or even necessary, in certain cases. Some cases which are prohibited by the ANSI standard generate a warning in this mode.

ANSI C Extensions

The extensions provided in this mode are as follows:

- Bitfields may have base types that are enumerated or integral types besides `int` and `unsigned int`. This matches A.6.5.8 in the ANSI Common Extensions appendix.

- Empty source files.
- A translation unit (input file) can contain no declarations.
- The last member of a structure may have an incomplete array type. It may not be the only member of the structure (otherwise, the structure would have zero size).
- A file-scope array can have an incomplete structure, union, or enumerated type as its element type. The type must be completed before the array is subscripted (if it is), and by the end of the compilation if the array is not `extern`.
- `enum` tags may be forward-declared; one may define the tag name and resolve it later (by specifying the brace-enclosed list).
- The values of enumeration constants may be given by expressions that evaluate to unsigned quantities that fit in the `unsigned int` range, but not in the `int` range. A warning is issued for suspicious cases. For example:

```
/* when ints are 32 bits: */
enum a { w = -2147483648}; /* No warning */
enum b { x = 0x80000000}; /* No warning */
enum c { y = 0x80000001}; /* No warning */
enum d { z = 2147483649}; /* Warning */
```

- An extra comma is allowed at the end of an `enum` list.
- A label or `case` label can be immediately followed by a right brace. (Normally, a statement must follow a label.)
- An initializer expression that is a single value and is used to initialize an entire static array, structure, or union need not be enclosed in braces. Strict ANSI C requires the braces.
- In an initializer with a file-scope variable, a pointer constant value may be cast to an integral type if the integral type is big enough to contain it.
- The address of a variable with `register` storage class can be taken. A warning is issued.
- In an integral constant expression, an integer constant may be cast to a pointer type and then back to an integral type.
- `long float` is accepted as a synonym for `double`.
- The `long long` and `unsigned long long` types are accepted. Integer constants suffixed by `LL` are given the type `long long`, and those

suffixes by ULL are given the type `unsigned long long` (any of the suffix letters may be written in lowercase).

- Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name may be redeclared in the same scope as long as it is declared with the same type. A warning is issued.
- Dollar signs (\$) are accepted in identifiers, even in strict dialects.
- Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token.
- Assignment and pointer difference are allowed between pointers to types that are interchangeable but not identical (for example, `unsigned char *` and `char *`). This includes pointer to same-sized integral types (for example, typically, `int *` and `long *`). Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, `int **` to `const int **`). Comparison and pointer difference of such pairs of pointer types are also allowed. A warning is issued.
- The expressions `p+i`, `p-i`, and `p-q` are allowed when `p` and `q` are of type `void *`, and behave like the same operations on `char *`. The expression `*p` is allowed, and produces a value of type `void`.
- Pointers to different function types may be assigned or compared for equality (`==`) or inequality (`!=`) without an explicit type cast. A warning is issued.
- A pointer to `void` may be implicitly converted to or from a pointer to a function type.
- The `#assert` preprocessing extensions of AT&T System V Release 4 are allowed. This feature is deprecated and may be removed in future versions of MULTI. These allow definition and testing of predicate names. Such names are in a name space distinct from all other names, including macro names. A predicate name is given a definition by a preprocessing directive of the form:

```
#assert name  
#assert name(token-sequence)
```

which defines the predicate *name*. In the first form, the predicate is not given a value. In the second form, it is given the value `token-sequence`. Such a predicate can be tested in a `#if` expression, as follows:

```
#name (token-sequence)
```

which has the value 1 if `#assert` of that name with that token-sequence has appeared, and 0 otherwise. A given predicate can be given more than one value at a given time. A predicate may be deleted by a preprocessing directive of the form:

```
#unassert name  
#unassert name (token-sequence)
```

The first form removes all definitions of the indicated predicate name; the second form removes just the indicated definition, leaving any others that might exist.

- The keyword `asm` is recognized as a synonym for `__asm`. This feature is disabled in Strict ANSI C mode because it conflicts with the ANSI C Standard for statements such as:

```
asm ("xyz") ;
```

ANSI C interprets this syntax as a call of an implicitly defined function `asm`, which (by default) the compiler interprets as an `asm` statement.

- `asm` macros are accepted, and `__asm` is recognized as a synonym for `asm`. An `asm` macro must be declared with no storage class and with a prototyped parameter list:

```
asm void f(int, int) {  
    ...  
}
```

For more information, see Chapter 19, “Enhanced `asm` Macro Facility for C and C++”.

- An extension is supported to allow constructs similar to C++ anonymous unions. In addition, anonymous structures are allowed—that is, their members are promoted to the scope of the containing structure and looked up in the same manner as ordinary members.
- An ellipsis can appear as the only thing in a function’s parameter list:

```
void f(...);
```

- External entities declared in other scopes are visible. A warning is issued. For example:

```
void f1(void) { extern void f(); }  
void f2() { f(); /* Using out of scope declaration */ }
```

- C99-style comments (using `/**` as delimiter) are supported.
- The C99 `_Pragma` preprocessing operator is accepted in all dialects.
- A structure that has no named fields, but at least one unnamed field (such as a zero-length bitfield), is accepted by default. A warning is issued.
- The `#warning` preprocessor directive, similar to `#error`, is supported in all dialects.
- Attributes, introduced by the keyword `__attribute__`, can be used on declarations of variables, functions, types, and fields. This extension mimics the GNU compiler for the attributes that are accepted. The following attributes are accepted:

- `alias`, `aligned`, `cdecl`, `common`, `const`, `deprecated`, `format`, `format_arg`, `no_check_memory_usage`, `no_instrument_function`, `nocommon`, `noreturn`, `packed`, `pure`, `section`, `stdcall`, `transparent_union`, `unused`, `used`, `volatile`, `weak`

- The `externally_visible` attribute expresses that a symbol might be referenced from outside of the program, and affects Wholeprogram optimizations. Similarly, you can specify that a symbol is externally visible using the `-external=` and `-external_file=` options (see “Optimization Scope” on page 179), or by using `#pragma ghs start_externally_visible` and `#pragma ghs end_externally_visible` (see “Green Hills Extension Pragma Directives” on page 683). To learn more about the use of the `externally_visible` attribute, see “Wholeprogram Optimizations” on page 333.
- The `__alignof__` keyword is supported. It is similar to `sizeof`, but returns the alignment requirement value for a type, or 1 if there is no alignment requirement. It can be followed by a type or expression in parentheses:

```
__alignof__(type)  
__alignof__(expression)
```

The expression in the second form is not evaluated.

- Preprocessing directive `#import` is supported.

```
#import "filename"
```

or

```
#import <filename>
```

causes *filename* to be included only once. This feature is deprecated and may be removed in future versions of MULTI.

- The preprocessor directive `#include_next` is recognized. This directive behaves in a similar manner to `#include`, but only searches those include directories specified on the command line after the present directory. This feature is deprecated and may be removed in future versions of MULTI.
- Keyword `__inline__` is supported (see “Manual Inlining In C” on page 318).
- Keyword `__typeof__` is supported. It takes an expression as the argument, and returns the type of the expression. For instance, the following code:

```
int i;  
__typeof__(i) j;
```

would declare variables `i` and `j`, both of type `int`.

- The macros `__FUNCTION__` and `__PRETTY_FUNCTION__` are supported. They expand to the name of the function surrounding them or, in the global scope, to the empty string. In C++, `__PRETTY_FUNCTION__` expands to the fully qualified name of the surrounding function.
- A function can be declared with the `__linkonce` storage class. The compiler will place the function in a special section whose name begins with `.ghs.linkonce` and whose name encodes the function name. The linker arbitrarily selects one object file from which to include a link-once section corresponding to each unique name. `__linkonce` functions cannot be static.
- The language is extended with the `__packed`, `__bytereverse`, `__bigendian`, and `__littleendian` type qualifiers. For more information about these type qualifiers, see “Type Qualifiers” on page 588.
- The language is extended with the `__thread` keyword, which is integrated with Cafe OS, see “Thread-Local Storage for Cafe OS” on page 590.
- The C99 `__Bool` type is accepted in all dialects.

- C99 digraph tokens, such as <% and %:, are accepted in all C and C++ dialects.
- As in C99, the C preprocessor treats all integer types as equivalent to `intmax_t` or `uintmax_t` during expression evaluation. For example:

```
int main()
{
    #if 0x80000000*2 != 0
        puts("The preprocessor uses 64-bit arithmetic on most targets");
    #endif
    if (0x80000000*2 == 0)
        puts("Normal arithmetic uses 32-bit arithmetic on 32-bit targets");
}
```

Strict ANSI C

-ANSI implements the standard C language that is defined in *X3.159–1989*. In this mode, only a few extensions to the standard are permitted (for example, `long long`) and a number of rules are strictly enforced.

To specify use of this dialect:



Set the **C/C++ Compiler→C Language Dialect** option to **Strict ANSI C (-ANSI)**.

GNU C

The GNU C dialect provides close compatibility with code written for the GNU compiler. It accepts all of the ANSI C dialect features, together with a number of GNU-specific extensions listed below. It is recommended that you use this dialect only when porting existing GNU-compiled code to the Green Hills development environment.

To specify use of this dialect:



Set the **C/C++ Compiler→C Language Dialect** option to **GNU C (-gcc)**.

GNU C Only Extensions

This dialect enables all the ANSI extensions listed in “ANSI C Extensions” on page 565, together with the extensions listed in this section, and those listed in “GNU C/C++ Extensions” on page 576.

- Variable length arrays (VLAs) are supported. GNU C also allows VLA types for fields of local structures, which can lead to run-time dependent sizes and offsets. The front end does not implement this, but instead treats such arrays as having length zero (with a warning); this enables some popular programming idioms involving fields with VLA types.

```
void f(int n)
{
    struct {
        int a[n]; /* Warning: variable-length array field type will be
                   treated as zero-length array field type */
    };
}
```

- A nonconstant may appear in the initializer list of an aggregate variable with automatic storage class.
- You can use GNU C-style inlining by using the keywords `inline` or `__inline__`. For more information about inlining functions manually, see “Manual Inlining” on page 318.

- Structs and unions without fields are accepted. They have size zero. Class types containing only fields of size zero (zero-length bitfields, zero-length arrays, and class types of size zero) also have size zero.
- In conditional expressions, the middle operand can be left out. When that is the case, the result of the expression is the value of the first operand if that first operand is “true”.

```
x = x ?: -1; /* If x is zero, evaluates to -1; otherwise, just x */
```

- Old-style definitions with unpromoted parameter types can follow prototype declarations with those same types.

```
void f(short);          // Prototype
void f() short p; {}    // Old-style definition OK
```

- An unprototyped declaration can follow a prototyped declaration. After such a redeclaration the original prototype is ignored. A warning is issued.

```
void f(char);           // Original prototype
void f();               // Unprototyped redeclaration
void f(double x) {}     // Accepted in GNU C mode
                        // (original prototype ignored)
```

- **GNU 3.3 ONLY** Certain “generalized lvalues” are supported. The ternary conditional operator produces an lvalue if its second and third operand are lvalues of the same type. Similarly, the comma operator produces an lvalue if its second operand is an lvalue. Some casts also preserve the lvalueness of their operand. For example:

```
unsigned x, y, z;
(x ? y : z) = 4; // Updates y or z depending on x (GNU 3.3 only)
(x += y, y) = 3; // Updates y (allowed in all non-strict modes)
(int)x = -1;     // Updates x (warning in all non-strict modes)
```

A “?” operator can also be treated as an lvalue if its operands have types that are different but are close enough that an lvalue cast can bridge the gap (for example, `int` and `int *`). A warning is issued.

- An expression can be cast to a union type containing a field with the type of that expression. For example:

```
union X { int i; char c; };
union X f(int i) { return (union X)i; }
```

- Implicit conversions are allowed between integral and pointer types, and between incompatible pointer types, with a warning. Implicit conversions between pointer types can drop cv-qualifiers (for example, `const char *` to `char *`).
- Pointer arithmetic applies to `void` pointers and function pointers as if they were `char *`.
- Explicit casts to struct and union types are allowed if they are do-nothing casts, that is, if the source type is the same as the destination type. (cv-qualifier differences are allowed but ignored.) The result is an lvalue if the source is an lvalue.
- Functions with return type `void` can have return expressions. A warning is issued if the return expression does not have type `void`.
- **GNU 3.3 ONLY** Bitfield lengths that are too large for the associated type are ignored with a warning (that is, the field becomes an ordinary field whose address can be taken). For example:

```
struct S {  
    char c: 9; // Oversized bitfield becomes regular field of type char  
};           // (assuming a char can hold at most 8 bits)
```

- **GNU 3.3 ONLY** Casts on an lvalue that do not fall under the usual “lvalue cast” interpretation (for example, because they cast to a type having a different size) are ignored, and the operand remains an lvalue. A warning is issued.

```
int i;  
(short)i = 0; // Accepted, cast is ignored; entire int is set
```

- “?” operators with mixed void/non-void operands in the second and third operands are accepted. The result has type `void`.
- **GNU 3.3 ONLY** In function definitions, local declarations can hide parameters.

```
void f(int x) {  
    double x; // Accepted in GNU C mode (with a warning).  
}
```

- An old-style parameter list can be followed by an ellipsis to indicate that the function accepts variable-length argument lists.

```
void f(x) int x; ... {} // Accepted in GNU C mode
```

GNU C/C++ Extensions

The extensions listed in this section are enabled when GNU C or GNU C++ is selected. See also “GNU C Only Extensions” on page 573 and “GNU C++ Extensions” on page 631.

- A function can be declared `inline`. It denotes that the inliner should attempt to inline calls to the function. The keyword `inline` is ignored (with a warning) on variable declarations and on block-extern function declarations.
- Keyword `typeof` is supported. It behaves exactly like `__typeof__` under safe GNU extensions.
- Implicit conversion of pointer types with the source type having a more qualified base type is allowed with a warning. Hence, only a warning is issued in the following case:

```
const char *s;  
char      *p;  
/* ... */  
p = s;                /* error in ANSI, warning with GNU */
```

- Compound literals are accepted. In addition to the features of C99 compound literals, GNU C mode also allows such literals to be used to initialize variables in file scope (if they only involve constant-expressions) and to initialize elements of aggregate types in brace-enclosed aggregate initializers.
- The `__extension__` keyword is accepted preceding declarations and certain expressions. It has no effect on the meaning of a program:

```
__extension__ __inline__ int f(int a) {  
    return a > 0 ? a/2 : f(__extension__ 1-a); }  
}
```

- Zero-length array types are supported. These are complete types of size zero.
- C99-style flexible array members are accepted. In GNU C++ mode, flexible array members are treated exactly like zero-length arrays.
- The `sizeof` operator is applicable to `void` and to function types and evaluates to the value 1.

- The “assembler name” of variables and routines can be specified. For example:

```
int counter __asm__("counter_v1") = 0;
```

Assembler names containing whitespace, non-identifier characters such as \$ or @, leading periods (.), or leading underscores (_) may not behave as expected.

- Register variables can be mapped onto specific registers using the `__asm__` keyword.

```
register int i __asm__("eax"); // Map "i" onto register eax.
```

- Excess aggregate initializers are ignored with a warning.

```
struct S { int a, b; };  
struct S a1 = { 1, 2, 3 }; // "3" ignored with a warning; no error  
int a2[2] = { 7, 8, 9 }; // "9" ignored with a warning; no error
```

- The `__restrict__` keyword is accepted. It is identical to the C99 `restrict` keyword, except for its spelling.
- Extended variadic macros are supported.
- C99 hexadecimal floating point constants are recognized.
- The `__asm__` keyword is recognized as a synonym for the `asm` keyword. Extended syntax is supported to indicate how assembly operands map to C/C++ variables.

```
asm("fsinx %1,%0" : "=f"(x) : "f"(a)); // Map the output operand on "x",  
                                         // and the input operand on "a".
```

- The `\e` escape sequence is recognized and stands for the ASCII “ESC” character.
- The address of a statement label can be taken by use of the prefix “&&” operator, for example, `void *a = &&L`. A transfer to the address of a label can be done by the “`goto *`” statement, for example, `goto *a`.
- ASCII NUL characters are accepted in source files.
- The last line of an include file can end with a backslash.
- Case ranges are supported. For example:

```
case 'a'...'z'
```

- A large number of special functions of the form `__builtin_xyz` (for example, `__builtin_alloca`) are predeclared.
- Some expressions are considered to be constant-expressions even though they are not so considered in standard C and C++. For example:
 - `((char *)&((struct S *)0)->c[0]) - (char *)0`
 - `(int)"Hello" & 0`
- The macro `__GNUC__` is predefined to the major version number of the emulated GNU compiler. Similarly, the macro `__GNUC_MINOR__` is predefined to the corresponding minor version number. Finally, `__VERSION__` is predefined to a string describing the compiler version.
- Nonstandard casts are allowed in null pointer constants. For example, the following is considered a null pointer constant in spite of the pointer cast in the middle:

`(int)(int *)0`
- Statement expressions such as `({int j; j = f(); j;})` are accepted. Branches into a statement expression are not allowed. In C++ mode, branches out are also not allowed. Variable-length arrays, destructible entities, try, catch, local non-POD class definitions, and dynamically initialized local static variables are not allowed inside a statement expression.
- Labels can be declared to be local in statement expressions by introducing them with a `__label__` declaration.

`({ __label__ lab; int i = 4; lab: i = 2*i-1; if (!(i%17)) goto lab; i; })`
- The C99 predefined identifier `__func__` is recognized.

The following GNU extensions are not currently supported in any GNU mode:

- GNU-style complex numbers (including complex literals).
- Nested functions.

Strict ISO C99

The Green Hills compiler and libraries fully support the ISO C99 standard, which introduced numerous features in both the language and libraries. Previous releases of the Green Hills compiler and library included some features of ISO C99 as extensions to ANSI C (in the **-ansi** dialect), but now the complete language and libraries are also provided as a separate dialect.

To use the Strict ISO C99 dialect:



Set the **C/C++ Compiler→C Language Dialect** option to **Strict ISO C99 (-C99)**.

ISO C99

The ISO C99 dialect provides all of the features of the language defined by *ISO/IEC 9899:1999*, in addition to several useful extensions. This mode is recommended when developing entirely new projects, particularly those that can benefit from the more precise definition of the language and extended features provided by C99. For a complete description of the ISO C99 language, see the *ISO/IEC 9899:1999* standard.

To use the ISO C99 dialect:



Set the **C/C++ Compiler→C Language Dialect** option to **ISO C99 (-c99)**.

In a few cases, the exact definition of a feature in the ISO C99 standard is different from the existing Green Hills implementation.

Features that are supported in C99 mode but not in ANSI mode:

- Complex types and **complex.h**.
- Hexadecimal floating point constants.
- The `restrict` keyword.
- Type-generic macros and **tgmath.h**

- Declarations after statements in a function or block, as in C++.
- Declaration of variables in the header of a `for` loop, as in C++.
- Non-constant initializers for automatic arrays, structures, and unions.
- `static` in parameter array declarators.
- Universal character names (UCNs) designated with `\u` or `\U`.

Preprocessor Support in C99 Mode

Preprocessor features that are only supported in C99 mode:

- The macro `__STDC_VERSION__` has the value `199901L` (in other modes the macro has the value `199409L`).
- `#pragma STDC FP_CONTRACT` and `#pragma STDC FENV_ACCESS`.
- `__func__` predefined identifier.
- The macros `__STDC_IEC_559__`, `__STDC_IEC_559_COMPLEX__`, and `__STDC_ISO_10646__` are undefined, which indicates that the implementation does not conform to the IEC-559 standard for floating point or the ISO-10646 standard for `wchar_t`.

Enforced Requirements in C99 Mode

Requirements that are enforced in C99 mode:

- Functions must be declared before use, although the prototype form is not required.
- Return statements must return a value if the function is non-void, and cannot return a value if the function is void.
- Function declarations cannot omit the return type or the type of parameters.

Different Features in C99 Mode

Features that are different in C99 mode:

- Large integer constants promote differently unless designated as `unsigned` or `long long` by using suffixes such as `U` or `LL`.

- Inline functions have different semantics (see “Manual Inlining In C” on page 318).
- Function-like macros in **math.h** (for example, `isinf()` and `isgreater()`) are implemented differently.
- Aliasing rules are more precise.
- New `struct` type compatibility rules (tag compatibility).

K&R C

Support for K&R C is deprecated and may be removed in future versions of MULTI.

K&R is the classic dialect of C in which UNIX was originally written. The first edition of *The C Programming Language* by Kernighan and Ritchie describes the K&R dialect. The most important implementation of this dialect was the *Portable C Compiler* (PCC). Although this dialect has important historical significance and might be necessary for compiling legacy applications, it is recommended that wherever possible, applications should be converted to one of the newer modes of the C language.

To specify use of this dialect:



Set the **C/C++ Compiler→C Language Dialect** option to **K&R C (-k+r)**.

The following is a list of incompatibilities between the Green Hills K&R Mode and PCC:

- Token pasting is not performed outside of macro expansions when only a comment separates two tokens. For example, `a/**/b` is not considered to be `ab`. The PCC behavior in that case can be achieved by preprocessing to a text file and then compiling that file.

Note that `/**/` may be used to concatenate two strings in macro definitions.

- PCC considers the result of a `? :` operator to be an lvalue if the first operand is constant and the second and third operands are compatible lvalues. The Green Hills compilers do not.
- PCC incorrectly parses the third operand of a `? :` operator in a way that some programs exploit. For example:

```
i ? j : k += 1
```

is parsed by PCC as

```
i ? j : (k += 1)
```

which is incorrect, since the precedence of `+=` is lower than the precedence of `? :`. The compiler will generate an error for that case.

The following section lists K&R Mode extensions to K&R C.

K&R Mode Extensions to K&R C

- the `void` keyword
- `enum` types
- the predefined macro names `__LINE__`, `__FILE__`, `__TIME__`, and `__DATE__`
- `#error`, `#ident`, `#elif`, `#defined(id)` preprocessor directives
- functions that return `struct` types
- `asm(str)` statements
- initialized `extern` variables
- initialized variables of `union` types
- initialized automatic variables of `struct` and `array` types
- keywords `const`, `signed`, and `volatile` are recognized
- Declarations of the form

```
typedef some-type void;
```

are ignored.

- Assignment is allowed between pointers and integers, and between incompatible pointer types, without an explicit cast. A warning is issued.
- A field selection of the form `p->field` is allowed if `p` does not point to a structure or union that contains `field`. `p` must be a pointer or an integer. Likewise, `x.field` is allowed even if `x` is not a structure or union that contains `field`. `x` must be an lvalue. For both cases, all definitions of `field` as a field must have the same offset within their structure or union.
- Overflows detected while folding signed integer operations on constants cause warnings rather than errors.
- A warning will be issued for an `&` applied to an array. The type of such an operation is “address of array element” rather than “address of array”.

- For the shift operators `<<` and `>>`, the usual arithmetic conversions are done, the right operand is converted to `int`, and the result type is the type of the left operand. In ANSI C, the integral promotions are done on the two operands, and the result type is the type of the left operand. The effect of this difference is that, in K&R mode, a `long` shift count will force the shift to be done as `long`.
- When preprocessing output is generated, the line-identifying directives will have the K&R form instead of the ANSI form.
- String literals will not be shared. Identical string literals will cause multiple copies of the string to be allocated.
- `sizeof` may be applied to bitfields; the size is that of the underlying type (for example, `unsigned int`).
- `lvalues` cast to a type of the same size remain `lvalues`, except when they involve a floating-point conversion.
- When a function parameter list begins with a `typedef` identifier, the parameter list is considered prototyped only if the `typedef` identifier is followed by something other than a comma or right parenthesis:

```
typedef int t;
int f(t) {}      /* Old-style list */
int g(t x) {}    /* Prototyped list, parameter x of type t */
```

That is, function parameters are allowed to have the same names as `typedef` identifiers. In the normal ANSI mode, any parameter list that begins with a `typedef` identifier is considered prototyped, so the first example above would give an error.

- The names of functions and of external variables are always entered at the file scope.
- A function declared `static`, used, and never defined is treated as if its storage class were `extern`.
- A file-scope array that has an unspecified storage class and remains incomplete at the end of the compilation will be treated as if its storage class is `extern` (in ANSI mode, the number of elements is changed to 1, and the storage class remains unspecified).
- The empty declaration:

```
struct x;
```

will not hide an outer-scope declaration of the same tag.

- In a declaration of a member of a structure or union, no diagnostic is issued for omitting the declarator list; nevertheless, such a declaration has no effect on the layout. For example:

```
struct s {  
    char a;  
    int;  
    char b[2];  
} v;          /* sizeof(v) is 3 */
```

- Enumerated types are always given type `int`. In ANSI mode, and depending on the command line options, smaller integral types will be used if possible.
- No warning is generated for a storage specifier appearing in other than the first position in a list of specifiers (as in `int static`).
- `short`, `long`, and `unsigned` are treated as “adjectives” in type specifiers, and they may be used to modify a `typedef` type.
- A “plain” `char` is considered to be the same as either `signed char` or `unsigned char`, depending on the target and command line options. In ANSI C, “plain” `char` is a third type distinct from both `signed char` and `unsigned char`.
- Free-standing tag declarations are allowed in the parameter declaration list for a function with old-style parameters.
- `float` function parameters are promoted to `double` function parameters.
- `float` functions are promoted to `double` functions.
- Declaration specifiers are allowed to be completely omitted in declarations (ANSI C allows this only for function declarations). Thus,

```
i;
```

declares `i` as an `int` variable. A warning is issued.

- All `float` operations are done as `double`.
- `__STDC__` is left undefined.
- Extra spaces to prevent the pasting of easily confused adjacent tokens are not generated in textual preprocessing output.
- The first directory searched for include files is the directory containing the file containing the `#include` instead of the directory containing the primary source file.
- Trigraphs are not recognized.

- Comments are deleted entirely (instead of being replaced by one space) in preprocessing output.
- `0x` is accepted as a hexadecimal value `0`, with a warning.
- `1E+` is accepted as a floating-point constant with an exponent of `0`, and a warning is emitted.
- The compound assignment operators may be written as two tokens (for example, `+=` may be written as `+ =`).
- The digits `8` and `9` are allowed in octal constants.
- A warning, rather than an error, is issued for integer constants that are larger than can be accommodated in an `unsigned long`. The value is truncated to an acceptable number of low-order bits.
- The types of large integer constants are determined according to the K&R rules (they will not be `unsigned` in some cases where ANSI C would define them that way). Integer constants with apparent values larger than `LONG_MAX` are typed as `long` and are also marked as “non-arithmetic”, which suppresses some warnings when using them.
- The escape `\a` (alert) is not recognized in character and string constants.
- Macro expansion is performed differently. Arguments to macros are not macro-expanded before being inserted into the expansion of the macro. Any macro invocations in the argument text are expanded when the macro expansion is rescanned. With this method, macro recursion is both possible and checked for.
- Token pasting inside macro expansions is performed differently. End-of-token markers are not maintained, so tokens that abut after macro substitution may be parsed as a single token.
- Macro parameter names inside character and string constants are recognized and appropriately substituted.
- Macro invocations having too many arguments are flagged with a warning rather than an error. The extra arguments are ignored.
- Macro invocations having too few arguments are flagged with a warning rather than an error. A null string is used as the value of the missing parameters.
- Extra `#else` statements (after the first has appeared in an `#if` block) are ignored, and a warning is emitted.

- The promotion rules for integers are different; unsigned char and unsigned short are promoted to unsigned int.
- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.

Motor Industry Software Reliability Association (MISRA) Rules

We provide support for the Motor Industry Software Reliability Association (MISRA) rules. For more information, see “MISRA C 2004” on page 197.

Japanese Automotive C Extensions

We provide support for the Japanese Automotive C extensions to ANSI C, which are used by Japanese automobile manufacturers. For complete specifications, refer to the *C-Language Specification for Automotive Control (Proposal)* by Toyota Motor Corporation, July 29, 1993.

Japanese Automotive C generally conforms to the principles of ISO 9899, equivalent to the ANSI *X3.159–1989* standard, with the exception of the “Implementation-Defined Behavior” specification of Annex G.3, which it modifies and extends to support portability. These extensions conform to the “Common Extension” section, in Annex G.5.

To enable support for the Japanese Automotive C extensions:



Set the **C/C++ Compiler→C Japanese Automotive Extensions** option to **On (-japanese_automotive_c)**.

Enabling **C/C++ Compiler→C Japanese Automotive Extensions (-japanese_automotive_c)** implicitly modifies the following options:

- Sets **C/C++ Compiler→C Language Dialect** option to **ANSI C (-ansi)**.
- Sets **C/C++ Compiler→Data Types→Signedness of Char Type** to **Unsigned (--unsigned_chars)**
- Sets **C/C++ Compiler→Data Types→Signedness of Bitfields** to **Unsigned (--unsigned_fields)**

- Sets **C/C++ Compiler→Data Types→Use Smallest Type Possible for Enum** to **Off** (`--no_short_enum`)
- Sets **Compiler Diagnostics→C/C++ Messages→Asm Statements** to **Silent** (`--asm_silent`)

In addition, the Green Hills tools satisfy the following Japanese Automotive C extensions by default:

- `#pragma intvect`, `#pragma asm`, and `#pragma endasm` are supported.
- If `sizeof(void *) == sizeof(int)`, any pointer to an object may be converted to an `int` and then back to a pointer that compares as equal to the original pointer.
- Bitfields may have arbitrary base types (satisfied by the `-ansi` option).

For example, if you compile the following code with the **C/C++ Compiler→C Language Dialect** option set to **Strict ANSI C** (`-ANSI`) :

```
struct {  
    char b:3;  
} s;
```

the compiler generates the following errors:

```
"x.c", line 2: error #230-D: nonstandard type for a bit field  
    char b:3;  
    ^
```

Enabling **C/C++ Compiler→C Japanese Automotive Extensions** (`-japanese_automotive_c`) sets the language dialect to **ANSI C** (`-ansi`), which suppresses these errors.

Type Qualifiers

The following type qualifiers are available:

__bytereverse

The `__bytereverse` type qualifier is used to designate that the data is stored in the opposite endianness, and that special means must be used to access

it. The `__bytereverse`d qualifier should not be used for automatic variables or parameters, although automatic pointers to `bytereverse`d objects are allowed.



Note During a debugging session, variables defined with `__bytereverse`d are displayed with their bytes in reverse order.

Example 1. Using the `__bytereverse`d Qualifier

```
__bytereverse int x;
void foo(void)
{
    /* OK - automatic pointer to bytereverse data */
    __bytereverse int *p = &x;

    /* Not OK - __bytereverse should not apply to an automatic variable */
    int * __bytereverse p2;

    ...
}
```

`__bytereverse`d should be applied only to integral and pointer types wider than `char`. `__bytereverse`d variables should not be defined with an initializer.

`__bigendian` and `__littleendian`

Similarly, `__bigendian` and `__littleendian` can be used in place of `__bytereverse`d to indicate a big or little endian byte ordering on the data. It is recommended that you do not use the type qualifier unnecessarily because this might cause problems or performance degradation if the code is ever ported to a CPU of the opposite endianness.

`__packed`

Use the `__packed` type qualifier to designate that an object might be misaligned, so the program must access it using a series of smaller, aligned accesses. For example, if you use `__packed` when declaring a pointer, you can safely use that pointer to access a field in a packed structure (see “Structure Packing” on page 108).

volatile

When your **C/C++ Compiler**→**C Language Dialect** is set to any mode other than K&R, enabling any of the **Optimization**→**Optimization Strategy** settings (see “Optimization Options” on page 176) will implicitly enable the **Advanced**→**Optimization Options**→**Memory Optimization** option (**-OM**), which assumes that memory locations only change under the control of the compiler. This assumption is usually, but not always, true. Exceptions include memory which is updated by interrupt routines, and memory-mapped I/O.

This optimization allows the compiler sometimes to avoid and/or delay reads or writes to memory locations by maintaining a copy of the memory location in a register. This is generally much faster because reads and writes to registers are faster than reads and writes to memory.

You can use the `volatile` qualifier to disable this optimization for particular variables (indicating that they may change), and retain its use for all other variables.

const

The compiler gives a compile-time error for any attempt to modify an object declared `const`. This qualifier also provides the compiler with additional information for use in optimizations. Wherever the value of a `const` variable is visible, the optimizer makes full use of the fact that this variable is simply a named constant value, combining it with other constants at compile time, and performing other simplifications. Even when the value of a `const` is not visible, the optimizer can make use of the fact that the variable is invariant to re-sequence statements and instructions or to move them outside of loops.

When `const` is used with `volatile`, the compiler never replaces a use of the object with its value, even if the value of the object is visible.

Thread-Local Storage for Cafe OS

With Cafe OS, SDK 2.10.00 or later, the `__thread` keyword can be used as a storage specifier to declare variables with external or static linkage in thread-local storage. The keyword may be used in the same context as a standard storage-class specifier, which may also be present. Thread-local

storage support is integrated with Cafe OS, and thread-local variables may be exported and imported from RPLs.

Example 2. Using the `__thread` Keyword

```
/* Define a zero-initialized thread-local variable of pointer type */
__thread int * ptr;

/* Define a thread-local variable initialized to 42 with static linkage */
static __thread int x = 42;

/* Declare an external thread-local variable, which must be defined elsewhere
   in the program. */
extern __thread int y;
```

Each thread will have its own copy of the thread-local variables, and each copy will be initialized before first use in each thread in accordance with its definition. There are a few restrictions on how thread-local variables may be used. Thread-local variables must have static initializers (i.e., compile-time constants). They must not be weak symbols. This implies that they may not satisfy a weak reference, as that would imply a mismatch between the weak reference and the strong definition. Finally, MISRA C 2004 rules 8.7 and 8.10 might not be enforced for thread-local variables, see “MISRA C 2004” on page 197.

Thread-local variables can be accessed in the debugger, which will naturally access the copy associated with the currently selected thread. However, if the thread does not have thread-local storage containing the variable in question (which may happen, for example, if the thread does not actually use thread-local storage), then the debugger will access the master copy of the variable that is used for initialization of new threads.

Assignment and Comparisons on struct and union Types

The assignment operator (=), which is supported in all modes of C, can be used to assign a value of a structure or a union type to a variable of the same type.

If there are padding bytes between fields or members of a `struct` or `union` type due to memory alignment requirements, those holes cannot be accessed by means of the structure or union. Note also that global variables will always be initialized to zero, and therefore the holes will always be zero; local variables, however, may have random data in the holes. Therefore, two structures or unions with the same values for every field might not be equal when compared.

For structures or unions that will be compared, it is important to either have no holes in the memory representation or to explicitly set the “hole” bytes to zero via a call to `memset` before the comparison.

A structure or a union can be passed as an argument to a function without restriction. The structure or union is copied when it is passed, however, so passing a very large structure or union is much less efficient than passing a pointer. For this reason, we recommend that pointers be passed where possible.

Bitfields

Bitfields in C have a base type which determines the field’s alignment, maximum size, and whether the field is signed or unsigned.

ANSI C Limitations

The ANSI C standard requires that the base type of a bitfield be either `int`, `signed int`, or `unsigned int`. This restriction is enforced when the **C/C++ Compiler→C Language Dialect** option is set to **Strict ANSI C** (`-ANSI`), and is ignored without a warning for all other dialects.

Signedness of Bitfields

Bitfields defined without either a `signed` or `unsigned` qualifier are unsigned by default (this default varies between target processor families). To control the signedness of bitfields manually:



Use the **C/C++ Compiler→Data Types→Signedness of Bitfields** option (`--signed_fields/--unsigned_fields`).

For example, if you set this option to `--signed_fields`:

- `int x:2` is a signed bitfield
- `char c:2` has the same signedness as the `char` type (see “**Signedness of Char Type**” on page 225)
- `enum { MIN=-2, MAX=1 } e:2` is a signed bitfield

- `enum { MIN=0, MAX=1 } e:2` is a signed bitfield
- `enum { MIN=0, MAX=3 } e:2` is an unsigned bitfield

If you set this option to **--unsigned_fields**:

- `int x:2` is an unsigned bitfield
- `char c:2` is an unsigned bitfield
- `enum { MIN=-2, MAX=1 } e:2` is a signed bitfield
- `enum { MIN=0, MAX=1 } e:2` is an unsigned bitfield
- `enum { MIN=0, MAX=3 } e:2` is an unsigned bitfield

However, single-bit bitfields of integer type that are not explicitly declared signed are always unsigned. Regardless of the setting of **--signed_fields** or **--signed_chars**:

- `int x:1` is an unsigned bitfield
- `char c:1` is an unsigned bitfield

The choice of signed versus unsigned bitfields can also affect code efficiency. Unless the processor has special instructions for this purpose, it may require two or even three instructions to extract the bits of a signed field whenever that field is evaluated. This is because the bits must be sign-extended to the size of an `int`, which may require two arithmetic shifts.

The following example demonstrates another difficulty with signed bitfields:

```
int i;
struct {int x:2; } y;
y.x = 2;
i = y.x;
if (y.x > 0) func();
```

- If `x` is an unsigned field, `i` is assigned the value 2 and **func()** is called.
- If `x` is a signed field, `i` is assigned the value -2 and **func()** is not called. This is because the range represented by a signed field with two bits is from -2 to 1. The value 2 is out of range. Out-of-range assignments are detected at compile time if warning 69 is enabled, or at run time if the translation unit has been compiled with **-check=assignbound** (see “**Run-Time Error Checks**” on page 188).

Size and Alignment of Bitfields

There is a close relationship between the base type of a field and the space it occupies, which can be summarized in three ways.

First, an individual bitfield can never have more bits than its base type.

Second, a bitfield must always fit entirely within a memory location that could hold its base type. It cannot cross a boundary that a simple variable of that type cannot cross. Thus, a field of type `char` can be placed in any memory location, but cannot cross a byte boundary. For example:

```
struct {  
    char a:4;  
    char b:6;  
    char c:6;  
} w;
```

Variable `w` requires three bytes because fields `a` and `b` do not fit in a single byte. Therefore, four bits of padding are inserted before field `b`. Similarly, the fields `b` and `c` do not fit in a single byte, so two bits of padding are inserted before the field `c`. On the other hand, if we had:

```
struct {  
    short a:4;  
    short b:6;  
    short c:6;  
} x;
```

then variable `x` would require only two bytes because the total size of the three fields is less than or equal to the size of a `short`.

Third, the alignment of a bitfield is determined by its base type. Padding may be inserted so that the field and the structure containing it are properly aligned. In the examples above, `w` has the alignment of a `char`, which is usually one byte, and `x` has the alignment of a `short`, which is usually either one or two bytes. Neither `w` nor `x` would be any larger due to alignment. However, in the following example, `y` and `z` will have an extra byte of padding added if `short` is 2 byte aligned:


```
struct {  
    char byte;  
    short a:4; /* padding inserted before  
                'a' for alignment */  
  
    short b:6;  
    short c:6;  
} y;  
struct {  
    short a:4;  
    short b:6;  
    short c:6;  
    char byte; /* padding added after  
                'byte' for alignment */  
} z;
```

For information about the size and alignment of C and C++ data types, see “PowerPC Characteristics” on page 104.

Enumerated Types

An enum type is treated as one of the integral types. By default, the members of an enum type (enumerators) are treated as integral constants of type `int` or greater (in case any of the values of the enumerators are out of `int` range). To instruct the compiler to attempt to use the smallest possible predefined type (including unsigned types) that allows representation of all listed values:



Set the **C/C++ Compiler**→**Data Types**→**Use Smallest Type Possible for Enum** option to **On** (`--short_enum`).

The ANSI standard does not call for strict type checking with respect to enum types. In fact, a variable of enumerated type is considered equivalent to any other integral value. Most operations which are allowed on values of integral types are also allowed on enumerated types, including assignment of a variable or parameter of one enumerated type to a variable or parameter of another type.

Functions with Variable Arguments

Green Hills C and C++ compilers support functions with variable parameters. In ANSI C modes and C++, it is done using the `<stdarg.h>` facility. In K&R

C mode, this is done using the `<varargs.h>` facility (though both this mode and the facility are deprecated). The two implementations are different and incompatible. Take care to use the correct facility for the appropriate mode of C.

An important limitation of variable parameters is that the types `char`, `short`, and `float` are not supported. When a function with variable parameters is called, the caller promotes expressions of these types to `int` (for `char` and `short`) and `double` (for `float`).

The `<varargs.h>` Facility

The `<varargs.h>` facility is deprecated and may be removed in future versions of **MULTI**.

To use the `<varargs.h>` facility, you must perform the following steps:

1. The line `#include <varargs.h>` must appear before the first function definition.
2. The last parameter of a variable argument list function must be named `va_alist`.
3. The last parameter declaration of a variable argument list function must be `va_dcl`.
4. There must not be a semicolon (;) between `va_dcl` and the initial left brace ({) of the function.
5. There must be a variable declared in the function of type `va_list`.
6. The `<varargs.h>` facility must be initialized at the top of the function by passing the variable of type `va_list` to a call of the macro `va_start`.
7. To obtain the variable arguments to the function, in left-to-right order, the macro `va_arg` is invoked once for each argument. The first argument to the macro `va_arg` is the variable of type `va_list`. The second argument is the type of the current argument of the function. The `va_arg` macro returns the value of the current argument of the function.
8. The `<varargs.h>` facility must be terminated by passing the variable of type `va_list` to a call of the macro `va_end` at the end of the function.

Example 3. Using the <varargs.h> Facility

```
#include <varargs.h>
/* Return the sum of a variable number of "int" arguments */
Sum(n, va_alist)
int n;
va_dcl                                /* steps 3 and 4 */
{
    va_list params;                    /* step 5 */
    int ret = 0;
    va_start(params);                 /* step 6 */
    while (n-- > 0) {
        ret += va_arg(params,int);    /* step 7 */
    }
    va_end(params);                   /* step 8 */
    return(ret);
}
```

The <stdarg.h> Facility

The <stdarg.h> facility has some additional limitations. First, every function with variable parameters is required to have at least one fixed parameter. Second, a prototype for the function must appear before its first invocation.

To use the <stdarg.h> facility, you must perform the following steps:

1. The line `#include <stdarg.h>` must appear before the first function definition.
2. In the function definition, at least one fixed parameter must appear in the parameter list before the ellipsis (`...`).
3. The last parameter in the function's parameter list must be ellipsis (`...`).
4. There must be a variable declared in the function of type `va_list`.
5. The variable argument processing must be initialized by invoking the macro `va_start` at the beginning of the function with two parameters: (1) the variable of type `va_list`, and (2) the last fixed parameter in the function parameter list.
6. To obtain the variable parameters to the function in left to right order, invoke the macro `va_arg` once for each parameter. The first parameter to `va_arg` is the variable of type `va_list`. The second parameter is the type of the current parameter of the function. The `va_arg` macro returns the value of the current parameter of the function.

7. The variable argument processing must be terminated by invoking the macro `va_end` with the variable of type `va_list` as its parameter. If the function passes the `va_list` variable to another function such as `vprintf`, the calling function must invoke `va_end` before returning.

Example 4. Using the `<stdarg.h>` Facility

```
#include <stdarg.h>

/* Return the sum of the parameter, with the first
   parameter being a parameter type string. */
double sum(const char *key, ...)
{
    double ret = 0;
    va_list ap;
    va_start(ap, key);
    while (*key) {
        switch (*key++) {
            case 'i': ret += va_arg(ap, int); break;
            case 'l': ret += va_arg(ap, long); break;
            case 'd': ret += va_arg(ap, double); break;
            case 'p': ret += *(va_arg(ap, int*)); break;
        }
    }
    va_end(ap);
    return ret;
}
```

The `asm` Statement

The `asm` statement generates in-line assembly code; it can be used anywhere a statement can be used within a function and anywhere a declaration can be used outside of a function. There are two spellings: `__asm` and `asm`. `asm` is supported in both ISO C99 and ANSI C modes (**-c99** and **-ansi**), but not their strict counterparts (**-C99** and **-ANSI**). `__asm` is supported in all modes.

```
__asm ("assembler_string");
```

or

```
asm ("assembler_string");
```

The entire contents of the string will be passed through to the assembly language output file in the same position as it appears in the source file. If the

underlying assembler requires a space or tab before the opcode, this tab or space must appear in the string.

See also Chapter 19, “Enhanced asm Macro Facility for C and C++”.

To control the diagnostic messages associated with asm statements (667 and 1546):



Set the **Compiler Diagnostics**→**C/C++ Messages**→**Asm Statements** option to one of the following settings:

- **Errors** (--asm_errors)
- **Warnings** (--asm_warnings)
- **Silent** (--asm_silent)



Note The compiler uses assembly language instructions and directives throughout the file without concern for possible interactions with user asm statements. Furthermore, the allocation of variables to registers and memory may vary from one compilation to another. Therefore, the use of the asm statement is considered non-portable and may be difficult to maintain.

The Preprocessor

Green Hills C and C++ compilers include a built-in preprocessor. Preprocessing is normally performed in a single pass. The behavior of the preprocessor is affected by the **C/C++ Compiler→C Language Dialect** option.

Preprocessor Output File

By default, preprocessing is performed concurrently with compilation and no intermediate output is generated. To write preprocessed output to a file:



When using the Project Manager, right-click the appropriate file and choose **Preprocess** from the context-sensitive menu.

When using the driver, pass the **-P** option.

Preprocessed output from a C or C++ file is saved with a **.i** extension, while output from an assembly file is saved with a **.si** extension.

You can also use the **-E** option to direct output to `stdout`. This latter option includes `#line` directives corresponding to the original file.

Extended Characters

English has a small alphabet, which is easily represented in eight bits by the ASCII character set. However, many languages require more space, and character sets that need more than 8 bits can be handled in two ways by ISO C/C++:

- *Multi-Byte Characters* — uses traditional C strings to represent the non-ASCII characters. For example, Kanji characters in the EUC or Shift-JIS format require two consecutive bytes to represent a single Kanji character. In both cases, the first byte has a value outside the normal ASCII range, which allows a program to recognize that the following byte should be treated as the rest of the current Kanji character. Multi-byte characters can be mixed with normal ASCII characters within the same string. Thus the length of a string in bytes does not directly correspond to the number of characters represented by the string. Some characters are one byte and others require more. Hence the term multi-byte.
- *Wide Characters* — are simply fixed-width (16- or 32- bit) characters. Wide characters were introduced by the ANSI/ISO C standard of 1990. The prefix

“L” is used to mark a character or string literal as wide. The `wchar_t` type matches the type of `L'x'` and `L"hello"[3]`.

The multi-byte string looks just like a traditional string in C, where each element of the string has type `char`. The wide character string uses more space because each element of the string has type `wchar_t`, which occupies either 16 or 32 bits depending on the target.

The advantages of wide characters over multi-byte characters are:

- The length of the string directly corresponds to the number of characters in that string. For example, the third character in a wide character string is always the third element, but in a multi-byte character string it could be one element or two. The character could even begin after the second, third, or fourth byte, depending upon whether there are Kanji or ASCII characters preceding it in the string.
- You can form a single wide character, such as `L'5'`, representing one multi-byte character.

However, wide characters take up more space and, more importantly, they cannot be manipulated by the large number of existing routines that were written for traditional C character strings, requiring instead special wide character functions.

Amendment 1 to the ISO C standard, adopted in 1995 and incorporated into both the ISO C++ standard and the ISO C99 standard, added many new functions to the standard C library in order to provide support for wide characters. These new functions, such as `wprintf()` and `wscanf()`, together with wide-character versions of most of the functions in `string.h` and `ctype.h`, have been added to **libansi.a**.



Note In addition to requiring the new functions, the standard also modifies the behavior of `printf()` and `scanf()` and their derivatives to support wide character strings and individual wide characters. The Green Hills implementation does not provide these modifications in order to minimize the impact of such changes on the size, speed, and stability of existing applications.

Compiler Support for Multi-Byte Characters

The Green Hills Software C/C++ compilers deal with multibyte character sets in two different places and reject them elsewhere. Multi-byte characters may

appear in comments and in character or string literals, but they may not appear in variable names.

The compiler may not need to know everything about a multibyte character set in order to parse comments or character and string literals which contain multi-byte characters. The compiler *does* need to be able to detect the end of a string or comment. Therefore, if the multibyte character set contains no values which have bytes that match punctuation the compiler is looking for, such as ' \ ", the compiler will be able to scan the comment or literal without requiring knowledge of the multibyte character set.

Parsing of wide character literals, however, is more difficult. The compiler must know the encoding of the multibyte character set in order to determine which bytes make up each element of type `wchar_t`. For example, `L"hello"` will result in a string of type `wchar_t[6]`, because each of the characters is ASCII (which the compiler understands). If, however, the five bytes in the string literal were not ASCII, the compiler must decide which bytes compose the first `wchar_t` element, and so on.

Applications requiring an unsupported multibyte character set will not be able to use wide character literals with the Green Hills Software C/C++ compilers. Instead, all literals must be written as multi-byte strings and converted using functions such as `mbstowcs()`. Furthermore, if the multibyte character set has bytes which match certain ASCII punctuation characters, multi-byte string literals may not be used either. In either case, it is possible to place multi-byte character strings in external data files and load them into the application at run time. The application can still make full use of the Green Hills C and C++ library routines to manipulate both wide character strings and multi-byte strings even if the compiler cannot parse string literals containing the multibyte character set.

The following list details the limitations to wide character support:

1. The regular `printf()` and `scanf()` functions do not support wide characters (such as `%lc` for `wint_t` or `%ls` for `wchar_t *`).
2. When using `wscanf()`, the behavior of the `%[]` modifier is defined only for characters in the range 0 to 0xffff.
3. Since the multibyte character sets that we support directly do not require state information, our implementation ignores `mbstate_t`. Consequently support for state-dependent encoding is not provided.

4. Locale is ignored. The default (and only) behavior is to support a common superset of EUC and Shift-JIS encodings.

Kanji Character Support

Kanji is the Japanese name for one of the written forms of Japanese. Almost all Kanji characters are taken from Chinese, although a few are specific to Japanese.

Both Chinese and Kanji use one or more characters to represent a word. In Chinese, each character is monosyllabic; in Japanese, a Kanji character can be monosyllabic, polysyllabic, or both, depending on the character. (Almost all Kanji characters have multiple pronunciations.) There are over 30,000 characters in Chinese, but Kanji routinely uses only a portion of them. For example, one can read a Japanese newspaper knowing fewer than 2,000 Kanji characters. The computer representations for Kanji provide between 5,000 and 10,000 Kanji characters.

In Japanese, there is also a phonetic alphabet with fewer than 100 characters. Just like the English alphabet, this alphabet has cursive and block forms. They are called Hiragana (cursive) and Katakana (block). Today Katakana characters are used to spell foreign words and proper names phonetically.

The Green Hills Software C/C++ compiler supports two representations of Kanji, EUC and Shift-JIS. To enable support for Kanji:



Set the **C/C++ Compiler**→**Special Tokens**→**Host & Target Character Encoding** option to:

- **EUC on Host and Target (-kanji=euc)**
- **Shift-JIS on Host and Target (-kanji=shiftjis)**
- **EUC on Host and Shift-JIS on Target (-kanji=euc/shiftjis)**

The Green Hills C and C++ compilers provide support for Kanji characters in comments, character strings, and character constants. Their use in variable names or other identifiers is not supported because of the conflicts it would cause with syntactic rules requiring identifiers to begin with a letter or underscore. For example, the EUC representation of Kanji uses two bytes for a single character, and neither byte is in the normal ASCII range. Thus a compiler can scan a C string literal containing EUC Kanji characters without any knowledge of the multi-byte representation. The Shift-JIS representation of Kanji, however, allows normal ASCII characters in the second byte. The

compiler must have knowledge of Shift-JIS in order to correctly parse a string literal containing Shift-JIS characters where the second byte has the value `\` or `"`, for example.

Some Japanese vendors provide a Kanji preprocessor which allows Kanji characters to be used in identifiers in C. Many companies manufacture PCs and Workstations with support for Kanji in both the keyboard and display.

Green Hills run-time libraries also correctly process character data containing Kanji characters. There are several different ways to represent Kanji characters. The Kanji representation directly supported by Green Hills compilers uses a pair of characters to represent a single Kanji character. The first character is always in the range `0xA0 – 0xFE` and the second character is always in the range `0x80 – 0xFE`.

The exact representation of Kanji is usually irrelevant to the compiler and libraries, as long as neither byte conflicts with special values such as `\0`, `\n`, `\r`, `"`, `\'`, etc.



Note In Shift-JIS mode, half-width Katakana characters in the range `0xA1 – 0xDF` are not supported by the wide character functions in the run-time library.

Compiler Limitations

The ANSI standard requires that a conforming compiler implementation accept programs of a certain complexity and size. These requirements are expressed in terms of a list of acceptable limits. The Green Hills compiler meets the requirements. For completeness, we list below all of the limits addressed by the ANSI standard, giving, first, the number which the standard requires and, second in parentheses, the number which the Green Hills compiler supports. The code (U) indicates the compiler is theoretically unlimited.

The code (U > *n*) indicates that the compiler is theoretically unlimited, and it is tested to handle at least the value *n*.

- 15 nesting levels of compound statements, iteration control structures and selection control structures (U > 500)
- 8 nesting levels of conditional inclusion (U > 500)
- 12 pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration (U > 64)

- 31 nesting levels of parenthesized declarators within a full declarator (U)
- 32 nesting levels of parenthesized expressions within a full expression (U > 200)
- 31 significant characters in an external identifier (U)
- 511 external identifiers in one translation unit (U)
- 127 identifiers with block scope declared in one block (U)
- 1024 macro identifiers simultaneously defined in 1 translation unit (U)
- 31 parameters in one function declaration (U > 100)
- 31 arguments in one function call (U > 100)
- 31 parameters in one macro definition (U > 100)
- 31 arguments in one macro call (U > 100)
- 509 characters in a logical source line (U > 3000)
- 509 characters in a character string literal or wide string literal (after concatenation) (U > 3000)
- 8 nesting levels for `#include` files (U > 500)
- 257 case labels for a `switch` statement, excluding those for any nested `switch` statements (U)
- 127 members in a single structure or union (U)
- 127 enumeration constants in a single enumeration (U)
- 15 levels of nested structure or union definitions in a single `struct-declaration-list` (U)

ANSI C Implementation-Defined Features

The ANSI standard for the C Programming Language (*X3J11/90-013*) leaves certain details of the language unspecified. Throughout the standard, various details are described as “undefined” or “implementation-defined”. The following definitions are taken from the standard for these and related terms:

Term	Meaning
implementation-defined behavior	Behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.
undefined behavior	Behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).
unspecified behavior	Behavior, for a correct program construct and correct data, for which this International Standard explicitly imposes no requirements.

The standard requires that a conforming implementation provide documentation which describes the behavior in each case where the standard uses the phrase “implementation-defined.” This chapter describes the behavior of the PowerPC compiler in each such case. Features are listed according to the sequence in which they appear in the standard, with citations referring to the appropriate section of the standard.

Translation J.3.1

Each non-empty sequence of whitespace characters, other than new line, is replaced by one space character.

Diagnostic messages all have the form:

```
"filename", line nnn: content_of_message  
                                offending_source_line
```

There are multiple classes of messages. For more information, see “Varying Message Severity” on page 264.

The compiler returns one (indicating failure), if any errors were reported, and zero (indicating success), if there were no errors.

The level of diagnostic can be controlled.

To suppress compiler warnings:



Set the **Compiler Diagnostics**→**Warnings** option to **Suppress (-w)**.

Environment J.3.2

No further processing occurs after program termination in a stand-alone environment.

In hosted execution environments, two arguments are passed to the function `main`:

```
int main(int argc, char *argv[]);
```

The argument `argc` is the number of valid elements in the `argv` array.

The argument `argv` is a null-terminated array of command line words.

In a stand-alone environment, no arguments are passed to the function `main`.

From the ISO C99 standard defined in *ISO/IEC 9899:1999*, "the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device" (section 7.19.3), and "what constitutes an interactive device is implementation-defined" (section 5.1.2.3). In a hosted environment that supports the `isatty()` system call, such as INTEGRITY, an interactive device is one for which `isatty()` returns a non-zero value. In a stand-alone environment, no streams are fully buffered until `setbuf()` or `setvbuf()` is called to change the buffering. For more information, see "Less Buffered I/O" on page 709.

Identifiers J.3.3

There is no limit to the number of significant initial characters in an identifier.

Case is significant in an identifier with external linkage.

Characters J.3.4

The escape sequence values are as follows.

Sequence	Value
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9
\v	11

There are no shift states in the encoding of multi-byte characters.

There are eight bits in a character in the execution character set.

The source and execution character sets are identical and the mapping is direct.

The basic execution character set includes all possible single-byte characters and the multibyte character set includes all possible 4-byte characters. Therefore, there are no characters or escape sequences represented in these two character sets.

An integer character constant may contain up to four characters. The value is calculated as follows: the least significant byte of the integer is the ASCII code for the rightmost character. The next byte of the integer is the ASCII code for the second character from the right, and so on.

A wide character constant that contains more than one multi-byte character has the same integer value as a wide character constant consisting only of the rightmost multi-byte character.

The C locale is used to convert multi-byte characters into corresponding wide characters (codes) for a wide character constant. The C locale is documented in *X3J11/90-013* under section 7.4.1.1 and in *ISO/IEC 9899:1999* under section 7.11.1.1.

The character type `char` is unsigned by default. To control the signedness of `chars` manually:



Use the **C/C++ Compiler**→**Data Types**→**Signedness of Char Type** option (**--signed_chars/--unsigned_chars**).

Integers J.3.5

The table below shows the storage and range of various variable types:

Designation	Size (bits)	Range
char	8	0..255
signed char	8	-128..127
unsigned char	8	0..255
short	16	-32768..32767
signed short	16	-32768..32767
unsigned short	16	0..65535
int	32	-2147483648..2147483647
signed int	32	-2147483648..2147483647
unsigned int	32	0..4294967295
long	32	-2147483648..2147483647
signed long	32	-2147483648..2147483647
unsigned long	32	0..4294967295

Truncation occurs when an integer is converted to a shorter signed integer.

Conversion of an unsigned integer to a signed integer of equal length does not modify the bit pattern of the number. Signed integers are represented in two's complement. Therefore such a conversion on a number with the high order bit set produces a negative number equal to *original_value* - 2^n , where n is the number of bits.

The result of bitwise operations on signed integers is a signed integer whose two's complement representation pattern is the result of applying the bitwise operation on the two's complement representation of the arguments.

The sign of the remainder of integer division will have the same sign as the dividend, if the remainder is not zero.

The right shift of a negative-value signed integral type produces a signed integral value, arithmetically shifted to the right.

Floating-Point J.3.6

The amount of storage and the range of various floating-point numbers are shown below:

Designation	Size (bits)	Range in bits
float	32	1.1754943635e-38 to 3.4028235e+38
double	64	2.2250738585072015e-308 to 1.7976931348623158e+308
long double	64	2.2250738585072015e-308 to 1.7976931348623158e+308

The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value is to the nearest representable value. In other words, the value is rounded.

The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number is to the nearest representable value. In other words, the value is rounded.

Arrays and Pointers J.3.7

The type of the integer required to hold the maximum size of an array is `unsigned int`. This is the type of `size_t`.

The type of integer required for a pointer to be converted to an integral type is `int`; its size is 4 bytes.

Casting a pointer to an `int` or vice-versa has no effect on the bit pattern of the value.

The type of the integer required to hold the difference between two pointers to members of the same array is `int`. This is the type of `ptrdiff_t`.

Registers J.3.8

The register storage class specifier causes the compiler to give first priority to the given variable when allocating variables to registers. Therefore, the register storage class specifier significantly affects allocation of variables. Because the compiler's automatic allocation of registers is very good, the register storage class specifier usually has a negative impact on allocation and should be used very sparingly.

Structures, Unions, Enumerations and Bit-fields J.3.9

Members of structures are padded and aligned as described in "PowerPC Characteristics" on page 104. All structure members except bitfields begin on a byte boundary. A structure is aligned according to the strictest alignment of any of its members. An array member is aligned according to the alignment requirement of its elements. Padding is performed when necessary to ensure that a member or bitfield begins on its required alignment. There is never any padding before the first member or bitfield, but there is padding after the last member or bitfield if it is needed to ensure that the size of the structure is a multiple of its alignment.

Bitfields may be declared with any standard integer base type (including signed long long and unsigned long long if supported). They may also be declared with `_Bool` or enum base types. When a bitfield is used in an integral context, its value is converted to a standard integer type in the following way. All standard integer types with rank greater than or equal to `int` are ordered from least to greatest rank with signed types preceding unsigned types of the same rank. The value is then converted to the first type in this ordering that can represent all of the values of the original bitfield type. For example, on a target where the `long` type has the same number of bits as the `int` type and the `long long` type is supported, a bitfield value would be promoted as follows. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, if an unsigned `int` can represent all values of the original type, the value is converted to an unsigned `int`; otherwise, if a `long long` can represent all values of the original type, the value is converted to a `long long`; otherwise, it is converted to an unsigned `long long`.

A non-packed bitfield must be wholly contained within a storage unit meeting the size and alignment requirements of its declared base type. Padding is inserted between bitfields as required to satisfy this constraint. For example:

```
struct {  
    int      a:20;  
    int      b:20;  
    long long c:20;  
};
```

On a target with 32-bit `ints` with 32-bit alignment, there will be 12 bits of padding between `a` and `b` since `b` cannot cross a 32-bit aligned boundary. On a target with 64-bit `long longs` with 64-bit alignment, there will also be 12 bits of padding between `b` and `c` since `c` cannot cross a 64-bit aligned boundary. On a target with 64-bit `long longs` with 32-bit alignment, there will be no padding between `b` and `c` since a `long long` can cross one 32-bit aligned boundary; however, if `c` were declared to contain more than 44 bits, there would then be 12 bits of padding between `b` and `c` since a 32-bit aligned `long long` cannot cross two 32-bit aligned boundaries.

As long as the above constraint is met, multiple bitfields, even those of different base types, may occupy the same storage unit. Within a particular storage unit, bitfields are allocated in increasing memory address order. This means on a big endian target the first bitfield will occupy the most significant bits of the storage unit, and on a little endian target the first bitfield will occupy the least significant bits. For example:

```
struct {  
    int    a:10;  
    char   b:4;  
    short  c:5;  
};
```

Assuming that `ints` are 32 bits, `shorts` are 16 bits, `chars` are 8 bits, and the alignment of each type matches its size, there will be no padding between `a` and `b`, but there will be 2 bits of padding between `b` and `c`. If bytes are 8 bits, all three fields will occupy one 4-byte storage unit, with `a` occupying the first byte, `a` and `b` sharing the second byte, and `c` occupying the third byte.

For information about controlling the signedness of bitfields, see “Signedness of Bitfields” on page 592. For information about controlling the size and type of enumerations, see “Enumerated Types” on page 595.

Qualifiers J.3.10

A volatile access occurs when a volatile object's value is either read or modified. If the volatile object is in memory, the compiler generates load and store instructions for each access. The compiler might still optimize away an access to an automatic (local) variable. As required by the standard, though, on return from `setjmp`, an automatic volatile variable has the value it had as of the corresponding call to `longjmp`.

Declarators J.3.11

There is no specific limit on the number of declarators that may modify an arithmetic, structure, or union type.

Statements J.3.12

There is no specific limit on the number of case values in a switch.

Preprocessing Directives J.3.13

The value of a single character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

Such a character constant can have a negative value.

If the file is included using the form `#include "file"`, the directory containing the file in which the `#include` directive occurs is searched first. If the directory does not contain the file, or if the file is using the form `#include <file>`, a list of user-specified directories is searched. For information about specifying include directories, see “Using Your Own Header Files and Libraries” on page 122.

In the `#include` directive, delimited character sequences are passed through to the operating system unchanged.

For a list of the recognized `#pragma` directives, see “Pragma Directives” on page 679.

The `__DATE__` and `__TIME__` macros always have a time available.

Library Functions J.3.14

NULL expands to `((void*) 0)` for the C language. For C++, the NULL macro expands to either 0 or 0L.

Given the following file **test.c**:

```
#include <stdio.h>
#include <assert.h>

int main()
{
    assert(1==2);
    return 0;
}
```

the program would print the following to `stderr` and then call `abort()`.

test.c, line 6: Assertion failed: "1==2"

The following table explains the return values of the listed functions for certain characters:

Function	Characters for which non-zero value is returned
isalnum()	'A'..'Z', 'a'..'z', and '0'..'9'
isalpha()	'A'..'Z' and 'a'..'z'
iscntrl()	0..31 and 127
islower()	'a'..'z'
isprint()	32..126
isupper()	'A'..'Z'

The following values are returned by math functions for domain errors:

Function(s)	Input values	Return value
acos(), acosf()	$ x > 1.0$	0.0
acosh()	$x < 1.0$	0.0
asin(), asinf()	$ x > 1.0$	0.0
asinh()	$x < 1.0$	0.0
atan2(), atan2f()	$x==0.0, y==0.0$	0.0
atanh()	$x==1$ or $x < -1$	0.0

Function(s)	Input values	Return value
fmod()	y == 0	0.0
log(), logf()	x <= 0.0	0.0
pow(), powf()	x == 0.0 and y < 0.0	0.0
sqrt(), sqrtf()	x < 0.0	0.0

Only the functions `exp()`, `pow()`, and `tan()` detect underflow and set `errno()` to `ERANGE`.

Locale-Specific Behavior(J.4)

Only the C locale is implemented. The C locale is documented in *X3J11/90-013* under section 7.4.1.1 and in *ISO/IEC 9899:1999* under section 7.11.1.1.

For a detailed description of the C locale's values related to `strftime()`, see *ISO/IEC 9899:1999*, section 7.23.3.4, paragraph 7.

The program is run in the local time zone and Daylight Savings Time.

The era for the `time()` function's Local Specific Behavior is Midnight, January 1, 1970 (GMT).

The characters in the execution set in addition to those required by the C standard are listed in "Characters J.3.4" on page 608.

The direction of printing is left to right.

The decimal point character is a period (.).

Character Testing and Case Mapping

The execution character set in its natural sequence is the ASCII character set.

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0A nl	0B vt	0C np	0D cr	0E so	0F si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 \$	25 %	26 &	27 '

28 (	29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C	7D }	7E ~	7F del

The `strftime()` function converts `%c`, `%x`, and `%X` as follows:

"%c"	is equivalent to	"%a %b %d %H:%M:%S %Y"
"%x"	is equivalent to	"%a %b %d, %Y"
"%X"	is equivalent to	"%H:%M:%S"

Additional Specifications

Signals

The equivalent of `signal(sig, SIG_DFL)` is executed prior to the call of a signal handler.

Default handling is reset if a `SIGILL` signal is received by a handler specific to the signal function.

Streams and Files

The last line of a text stream does not require a terminating new line character.

Space characters written out to a text stream immediately before a new line character appear when the stream is read back.

No null characters are appended to data written to a binary stream.

The file position indicator of an append mode stream is initially positioned at the end of the file.

A write on a text stream does cause the associated file to be truncated beyond that point.

Unbuffered, fully buffered, and line buffered modes are all provided in hosted environments. Embedded environments provide a limited form of buffering which is essentially unbuffered, but which provides the benefits of buffering within operations such as `fwrite()` and `printf()`.

A zero length file can exist.

The rules for composing a valid filename differ between systems.

The same file can be opened multiple times.

Removing an open file has unpredictable results. In most cases the application program will only have access to a portion of the file as it was before it was deleted. Once the application exits, the file will cease to exist entirely even if the application was writing to the file.

The effect of renaming a file to a name of a file which already exists varies between systems. On some systems it is undetected and the old file is lost. On others, the rename fails and all files remain unchanged.

The `%p` format descriptor in `printf()` behaves exactly as `%lx`.

The `%p` format descriptor in `scanf()` behaves exactly as `%x`.

If the `-` is neither the first character nor the last character in the scan list for `[` conversion in the `fscanf` function, all characters in the range beginning with the character preceding the `-` up to and including the character following the `-` are added to the scan set. If the character following the `-` lexically precedes the character before the `-`, then the `-` and the character following it are ignored.

tmpfile

An open temporary file is not removed if the program terminates abnormally.

errno

The functions `fgetpos()` and `ftell()` set `errno` to `EBADF` on failure.

The function `perror()` prints a message in this format:

argument: contents_of_message

where *argument* is the parameter passed to `perror` and *contents_of_message* is a predefined message associated with the current value of `errno`.

Memory

The functions `calloc()` and `malloc()` return a valid pointer if the size requested is zero.

When you request a size of zero with the `realloc()` function, it returns `NULL` if you pass it a valid pointer, or a valid pointer if you pass it `NULL`.

abort()

When `abort()` is called, the buffers within the **stdio** library for open files are not flushed. The files are closed and exist. Temporary files are not deleted.

The `abort()` function in the stand-alone library is equivalent to calling `raise(SIGABRT)` followed by `exit(EXIT_FAILURE)`. It does not call C++ destructors for global objects or functions registered with `atexit()`.

exit()

The low-order eight bits of the argument passed to `exit()` are returned as the status.

The `exit()` function in the stand-alone library is equivalent to calling the functions registered by `atexit()`, followed by `_Exit(status)`.

getenv()

The only limitation on environment names is that they may not contain the character `=` which is used to separate the name from its value.

The method to alter the environment list obtained by a call to the `getenv` function is to call the function `setenv(char *name, char *value)`. This will set the environment variable named by the null-terminated string `name` to a copy of the null-terminated string pointed to by `value`.

strerror()

The strings returned by `strerror` for the various values of `errno` are listed below:

00: "Error 0 (no error)"	18: "Cross-device link"
01: "Not owner"	19: "No such device"
02: "No such file or directory"	20: "Not a directory"
03: "No such process"	21: "Is a directory"
04: "Interrupted system call"	22: "Invalid argument"
05: "I/O error"	23: "File table overflow"
06: "No such device or address"	24: "Too many open files"
07: "Arg list too long"	25: "Not a typewriter"
08: "Exec format error"	26: "Text file busy"
09: "Bad file number"	27: "File too large"
10: "No children"	28: "No space left on device"
11: "No more processes"	29: "Illegal seek"
12: "Not enough core"	30: "Read-only file system"
13: "Permission denied"	31: "Too many links"
14: "Bad address"	32: "Broken pipe"
15: "Block device required"	33: "Argument too large"
16: "Mount device busy"	34: "Result too large"
17: "File exists"	35: "Non standard error"

There is little formatting style to the string returned by `strerror()`. The first character is always capitalized and the string does not end with a period.

Green Hills C++

Chapter 15

This Chapter Contains:

- Specifying a C++ Language Dialect
- Specifying C++ Libraries
- Standard C++
- Embedded C++
- Extended Embedded C++
- Features Supported by Each C++ Dialect
- Standard C++ with ARM Extensions
- GNU C++
- Template Instantiation
- Multiple and Virtual Inheritance
- Exception Handling
- Namespace Support
- Precompiled Header Files
- Linkage
- Post Processing in C++
- The C++ decode Utility
- C++ Implementation-Defined Features
- Deprecated C++ Headers

This chapter describes the Green Hills implementation of C++, including aspects of the language particular to our compilers.

The needs of C++ users vary widely, depending on a number of factors, including the type of target application and environment, the foreign libraries used, compatibility with other C++ compilers, and the trade-offs users make in regard to the C++ feature set and library support they require. To meet such a diverse set of needs, Green Hills Software supports a form of “scalable” C++. You can specify language and library levels to obtain everything from a small and efficient Embedded C++ compiler and library, to full ISO Standard C++.

Specifying a C++ Language Dialect

To specify a C++ dialect:



Set the **C/C++ Compiler**→**C++ Language Dialect** option to one of the available settings. For a list of driver options, see “**C++ Language Dialect**” on page 195.

Several dialects are available, depending on which OS you are using for your project:

- The *International Standard ISO/IEC 14882:1998* dialect (see “Standard C++” on page 624).
- The *Annotated C++ Reference Manual (ARM)* dialect (see “Standard C++ with ARM Extensions” on page 630).
- The GNU dialect (see “GNU C++” on page 631).
- Embedded C++ (see “Embedded C++” on page 627).
- Extended Embedded C++ (see “Extended Embedded C++” on page 629).



Note The Green Hills Tools do not support mixing C++ language dialects. Use the same dialect when compiling and linking all C++ files in your project.

Specifying C++ Libraries

Depending on how you set **C/C++ Compiler→C++ Language Dialect** and **C/C++ Compiler→C++ Exception Handling**, the Green Hills Tools use the following default for **C/C++ Compiler→C++ Libraries**:

C++ Language Dialect	C++ Exception Handling	Default C++ Libraries
Standard C++ --STD, --std, --arm, --g++	On --exceptions	C++ with exceptions --stdle
	Off --no_exceptions	C++ without exceptions --stdl
Extended Embedded C++ --ee	On --exceptions	EEC++ with exceptions --eele
	Off --no_exceptions	EEC++ without exceptions --eel
Embedded C++ --e	On --exceptions	EC++ with exceptions --ele
	Off --no_exceptions	EC++ without exceptions --el

To link against a different set of Green Hills libraries, set the **C/C++ Compiler→C++ Libraries** option manually. You must specify an option that is less full-featured than the default (and therefore lower in the table). The Green Hills Tools do not support mixing C++ libraries. Use the same library when compiling and linking all C++ files in your project.

For example, if you want to write code in C++ that uses exceptions while restricting your use of library modules to those offered in EEC++, use the following options:

```
--std --exceptions --eele
```

The **C/C++ Compiler→C++ Libraries** option also sets the default `#include` paths so that the tools use the correct header files for the selected libraries.



Note We recommend that you do not mix code that handles exceptions with libraries that do not handle exceptions, or vice versa, as you may get unexpected results when running your program.

If you do not want to link against any Green Hills standard libraries, see “**Link in Standard Libraries**” on page 310.

Standard C++

This dialect is defined by the *International Standard ISO/IEC 14882:1998* (except as noted in this chapter). It also includes additional features and changes developed after the standard. To specify use of this dialect:



Set the **C/C++ Compiler→C++ Language Dialect** option to either:

- **Standard C++ (Violations give Errors) (--STD)**, or
- **Standard C++ (Violations give Warnings) (--std)** [default]

Some of the features in the standard are disabled by default. In order to obtain the exact dialect specified by the standard, you need to set these additional options as follows:

- Set the **C/C++ Compiler→C++ Exception Handling** option to **On (--exceptions)**.
- Set the **C/C++ Compiler→C++→Templates→Recognition of Exported Templates** option to **On (--export)**.
- Set the **Advanced→C/C++ Compiler Options→Advanced C++ Options→Distinct C and C++ Functions** option to **On (--c_and_cpp_functions_are_distinct)**.
- Set the **C/C++ Compiler→C++→Keyword Support→Instantiate Extern Inline** option to **On (--instantiate_extern_inline)**.
- Set the **C/C++ Compiler→C++→Templates→Implicit Source File Inclusion** option to **Off (--no_implicit_include)**.

By default, the **--STD** option enables **C/C++ Compiler→C++→Templates→Dependent Name Processing (--dep_name)**. For more information about **--dep_name**, see “Namespace Support” on page 645.

Extensions Accepted in Normal C++ Mode

The following extensions are accepted in all modes (except when strict ANSI violations are diagnosed as errors):

- A friend declaration for a class may omit the `class` keyword:

```
class B;
class A {
    friend B;           // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes:

```
class A {
    const int size = 10;
    int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
    int A::f();         // Should be "int f()"
};
```

- “Anonymous classes” and “anonymous structures” are allowed as long as they have no C++ features (e.g., no static data members or member functions and no nonpublic members) and have no nested types other than other anonymous classes, structures, or unions. For example:

```
struct A {
    struct {
        int i, j;
    };                // Okay -- references to A::i and A::j
                      // are allowed
};
```

- The type qualifier `restrict` is allowed for reference and pointer-to-member types and for nonstatic member functions.
- Implicit type conversion between a pointer to an extern "C" function and a pointer to an extern "C++" function is permitted. For example:

```
extern "C" void f();   // f's type has
                      // extern "C" linkage
void (*pf)() = &f;    // pf points to an
                      // extern "C++" function
                      // error unless implicit
                      // conversion is allowed
```

For more information, see “**Distinct C and C++ Functions**” on page 308.

- In **--arm** mode, a `?` operator whose second and third operands are string literals or wide string literals can be implicitly converted to `char *` or `wchar_t *`, respectively. In C++, string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to `char *`, dropping to `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a `?` operation is an extension.

```
char *p = x ? "abc" : "def";
```



Note For more information about Standard C++, we recommend Stroustrup's *The C++ Programming Language, Third Edition*.

Embedded C++

The Embedded C++ dialect (EC++) is intended as a stable, simple, and efficient version of Standard C++, intended to produce an open, worldwide standard. EC++ is not a new language specification that competes with existing Standard C++; it is a pure subset for the practical user of C++. The EC++ library is smaller and more efficient than the Standard C++ library. For more information about EC++, visit the Embedded C++ Web site (<http://www.caravan.net/ec2plus/>). For a feature comparison with Standard C++, see “Features Supported by Each C++ Dialect” on page 629.

To specify use of this dialect:



Set the **C/C++ Compiler**→**C++ Language Dialect** option to **Embedded C++ (--e)**.

Differences Between Standard C++ and Embedded C++

In addition to the features and omissions mentioned in “Features Supported by Each C++ Dialect” on page 629, the following features behave differently in Embedded C++ than in Standard C++.

std::string

In C++, `string` is a typedef for:

```
basic_string<char, char_traits<char>, allocator<char> >
```

In EC++, none of these template classes exist, and `string` is provided as a non-template class. The EC++ library does not provide any wide character version of `string` (there is no `wstring`).

Complex Numbers

In C++, the `<complex>` header defines a template class `complex<T>` with specializations for `float`, `double`, and `long double`. In EC++, this template class does not exist, and the `<complex>` header defines two non-template classes `float_complex` and `double_complex`, with

all the same member functions as the standard `complex<float>` and `complex<double>`, respectively. The EC++ library does not provide any equivalent to `complex<long double>`.

Streams and Buffers

In C++, the standard library provides a large number of “stream” and “buffer” classes — `filebuf`, `streambuf`, `stringbuf`, `istream`, `ostream`, `ifstream`, and `ofstream` — each of which is implemented as a specialization of an underlying template class named `basic_class`. In EC++, none of these underlying template classes exist, but the EC++ library provides non-template versions of the stream classes with the same interfaces as the standard C++ versions. The EC++ library does not provide any wide character streams or buffers (there is no `wistream`, `wstringbuf`, etc.).

The standard C++ classes `iostream` and `strstream` require multiple inheritance, and therefore are not provided by EC++.

The only standard `iostream` objects provided by EC++ and EEC++ are `cin` and `cout`. The `cerr` and `clog` objects are omitted for simplicity, and the wide `iostream` objects `wcin`, `wcout`, `wcerr`, and `wclog` are omitted along with all of the wide stream and buffer types.

Extended Embedded C++

Extended Embedded C++ is a hybrid of Standard C++ and Embedded C++, created in a joint effort by Green Hills Software and P. J. Plauger. It is often referred to as EEC++ or ESTL (Embedded C++ with the Standard Template Library). It is not an official standard. For a feature comparison with Standard C++ and EC++, see “Features Supported by Each C++ Dialect” on page 629.

To specify use of this dialect:



Set the **C/C++ Compiler**→**C++ Language Dialect** option to **Extended Embedded C++ (--ee)**.

Features Supported by Each C++ Dialect

The following table compares the feature set of Standard C++, Extended Embedded C++ (EEC++), and Embedded C++ (EC++). Features that are not listed in this table can be individually controlled through Builder and driver options. For more information about features marked with EC++, see “Differences Between Standard C++ and Embedded C++” on page 627.

Feature	Standard	EEC++	EC++
Exception support	Optional	Optional	Optional
Complex numbers	C++	EC++	EC++
Streams and buffers	C++	EC++	EC++
string	C++	EC++	EC++
wchar_t	Yes	Yes	Yes
cin, cout	Yes	Yes	Yes
Template support	Yes	Yes	No
Standard container classes	Yes	Yes	No
Namespace support	Yes	Yes	No
std:: namespace	Yes	Yes	No
bitset<>	Yes	Yes	No
<algorithm>, <functional>, <memory>	Yes	Yes	No

Feature	Standard	EEC++	EC++
mutable keyword	Yes	Yes	No
static_cast<>, const_cast<>, reinterpret_cast<>	Yes	Yes	No
Multiple inheritance	Yes	No	No
dynamic_cast<>	Yes	No	No
<locale>	Yes	No	No
<limits>	Yes	No	No
wcin, wcout, etc.	Yes	No	No
cerr, clog	Yes	No	No
iostream, stringstream	Yes	No	No
<typeinfo>	Yes	No	No

Standard C++ with ARM Extensions

This dialect is ARM-compliant C++ as defined by *The Annotated C++ Reference Manual* (ARM) by Ellis and Stroustrup (Addison-Wesley, 1990), including templates, exceptions, and the anachronisms listed in Chapter 18 of that book. This is essentially the same language defined by the language reference for Cfront version 3.0x, with the addition of exceptions.

To specify use of this dialect:



Set the **C/C++ Compiler→C++ Language Dialect** option to **Standard C++ with ARM Extensions (--arm)**.



Note This dialect is not supported on INTEGRITY projects.

GNU C++

This dialect closely emulates the C++ language supported by the GNU compiler.

To specify use of this dialect:



Set the **C/C++ Compiler**→**C++ Language Dialect** option to **GNU C++** (**--g++**).



Note This dialect is not supported on INTEGRITY projects.

GNU C++ Extensions

This dialect enables the extensions listed below, together with those in “GNU C/C++ Extensions” on page 576.

- If an explicit instantiation is preceded by the keyword `extern`, no (explicit or implicit) instantiation is created for the indicated specialization.
- A special keyword `__null` expands to the same constant as the literal “0”, but is expected to be used as a null pointer constant.
- When dependent name processing is disabled, names from dependent base classes are ignored only if another name would be found by the lookup.

```
const int n = 0;
template <class T> struct B {
    static const int m = 1; static const int n = 2;
};
template <class T> struct D : B<T> {
    int f() { return m + n; } // B::m + ::n in g++ mode
};
```

- When doing name lookup in a base class, the injected class name of a template class is ignored.

```
namespace N {
    template <class T> struct A {};
}
struct A {
    int i;
};
struct B : N::A<int> {
    B() { A x; x.i = 1; } // g++ uses ::A, not N::A
};
```

- The injected class name is found in certain contexts in which the constructor should be found instead.

```
struct A {
    A(int) {};
};
A::A a(1);
```

- A difference in calling convention is ignored when redeclaring a typedef, even when **--c_and_cpp_functions_are_distinct** is specified.

```
typedef void F();
extern "C" {
    typedef void F(); // Accepted in GNU C++ mode (error otherwise)
}
```

- The macro `__GNUG__` is defined identically to `__GNUC__` (i.e., the major version number of the GNU compiler version that is being emulated).
- The macro `_GNU_SOURCE` is defined as “1”.
- Guiding declarations (a feature present in early drafts of the standard, but not in the final standard) are disabled.
- **GNU 3.3 ONLY** A friend declaration may refer to a member typedef.

```
class A {
    class B {};
    typedef B my_b;
    friend class my_b;
};
```

- An inherited type name can be used in a class definition and later redeclared as a typedef.

```
struct A { typedef int I; };
struct B : A {
    typedef I J;          // Refers to A::I
    typedef double I;     // Accepted in g++ mode
};                        // (introduces B::I)
```

- In a catch clause, an entity may be declared with the same name as the handler parameter.

```
try { }
catch(int e) {
    char e;
}
```

- A template argument list may appear following a constructor name in a constructor definition that appears outside of the class definition:

```
template <class T> struct A {
    A();
};
template <class T> A<T>::A<T>() {}
```

- **GNU 3.3 ONLY** A constructor need not provide an initializer for every nonstatic const data member (but a warning is still issued if such an initializer is missing).

```
struct S {
    int const ic;
    S() {} // Warning only in GNU C++ mode (error otherwise)
};
```

- Exception specifications are ignored on function definitions when support for exception handling is disabled (normally, they are only ignored on function declarations that are not definitions).
- A friend declaration in a class template may refer to an undeclared template.

```
template <class T> struct A {
    friend void f<>(A<T>);
}
```

- A function template default argument may be redeclared. A warning is issued and the default from the initial declaration is used.

```
template<class T> void f(int i = 1);  
template<class T> void f(int i = 2) {}  
int main() {  
    f<void>();  
}
```

- A definition of a member function of a class template that appears outside of the class may specify a default argument.

```
template <class T> struct A { void f(T); };  
template <class T> void A<T>::f(T value = T()) {}
```

Template Instantiation

The C++ language includes the concept of templates. A template is a description of a class or function that is a model for a family of related classes or functions. For example, you can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as *template entities*) are not necessarily done immediately, for several reasons:

- You would like to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)
- The language allows you to write a specialization of a template entity, that is, a specific version to be used in place of a version generated from the template for a specific data type. (You could, for example, write a version of `Stack<int>`, or of just `Stack<int>::push`, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) While compiling a reference to a template entity, the compiler cannot know if a specialization for that entity will be provided in another compilation. Thus, it cannot do the instantiation automatically in any source file that references it.

- The language also dictates that template functions that are not referenced should not be compiled, and that, in fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

It should be noted that certain template entities are always instantiated when used (e.g., inline functions).

The Green Hills C++ compiler provides two methods of template instantiation, which are described in the following.

Prelinker Template Instantiation

Prelinker instantiation, selected with the **--no_link_once_templates** option, is the default mode of template instantiation. To explicitly enable it:



Set the **C/C++ Compiler**→**C++**→**Templates**→**Link-Once Template Instantiation** option to **Off** (**--no_link_once_templates**). For more information about this option, see “Templates” on page 234.

The Green Hills C++ prelinker instantiation method works as follows:

1. The first time the source files of a program are compiled, no template entities are instantiated. However, template information files are generated and contain information about entities that could have been instantiated in each compilation. These template information files have a **.ti** suffix.
2. When the object files are linked together, a program called the *prelinker* is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.
3. If the prelinker finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. Upon finding such a file, the prelinker assigns the instantiation to it. The set of instantiations assigned to a given file is recorded in an associated instantiation request file (with a **.ii** suffix).
4. The prelinker then executes the compiler again to recompile each file for which the **.ii** file was changed. The original compilation options (saved in the **.ti** file) are used for recompilation.

5. When the compiler compiles a file, it reads the `.ii` file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and everything else already in the object file). The compiler also receives a definition list file, which lists all the instantiations for which definitions already exist in the set of object files. If during compilation the compiler has the opportunity to instantiate a referenced entity that is not on that list, it performs the instantiation. It passes back to the prelinker (in the definition list file) a list of instantiations that it has “adopted” in this way, so the prelinker can assign them to a file. This adoption process allows rapid instantiation and assignment of instantiations referenced from new instantiations, and reduces the need to recompile a given file more than once during the prelinking process.
6. The prelinker repeats steps 3-5 until there are no more instantiations to be adjusted.
7. The object files are linked together.

After the program has been linked correctly, the `.ii` files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the `.ii` files and do the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no adjustments will need to be made to the instantiation assignments. This is true even if the entire program is recompiled.

If the programmer provides a specialization of a template entity somewhere in the program, the specialization will be seen as a definition by the prelinker. Since that definition satisfies whatever references there might be to that entity, the prelinker will see no need to request an instantiation of the entity. If the programmer adds a specialization to a program that has previously been compiled, the prelinker will notice that too and remove the assignment of the instantiation from the proper `.ii` file.

The `.ii` files should not, in general, require any manual intervention. There are two exceptions.

1. If the following are true:
 - A definition is changed in such a way that some instantiation no longer compiles (it gets errors), and
 - A specialization is added in another file, and

- The first file is being recompiled before the specialization file and is getting errors,
You must delete the **.ii** file for the file getting the errors to allow the prelinker to regenerate it.
2. If you use multiple major versions of the tools to build your project (for example, MULTI 4 and MULTI 5), we recommend that you use multiple build directories. If you do not use multiple build directories, you might get errors when building your project in one release when **.ti** files reference configuration options that only exist in the other release. If you get these errors, remove the appropriate **.ti** and **.ii** files.

If the prelinker changes an instantiation assignment, it will issue a message such as:

```
C++ prelinker: A<int>::f() assigned to file test.o
```

Instantiations are normally generated as part of the object file of the translation unit in which the instantiations are performed. But when *one instantiation per object* is used, each instantiation is placed in its own object file. This mode is useful when building libraries that need to include copies of the instances referenced from the library. If each instance is not placed in its own object file, it may be impossible to link the library with another library containing some of the same instances.

To enable one instantiation per object:



Set the **C/C++ Compiler**→**C++**→**Templates**→**One Instantiation Per Object** option to **On** (**--one_instantiation_per_object**). For more information about this option, see “Templates” on page 234.



Note If you use one instantiation per object and you have a template that calls a static function, you might not get debug information for that function.

Link-Once Template Instantiation

An alternative method of template instantiation is known as link-once template instantiation. In this method, non-export templates that are used in a given source file are always instantiated during compilation. Because a template can be used by multiple source modules, some convention must be used to ensure that the definitions of such templates in multiple object files does not

result in multiply defined symbols at link time. The templates are instantiated into specially named link-once sections as if they had been declared with `__linkonce` (see “ANSI C Extensions” on page 565).

To enable link-once template instantiation:



Set the **C/C++ Compiler**→**C++**→**Templates**→**Link-Once Template Instantiation** option to **On** (`--link_once_templates`). For more information about this option, see “Templates” on page 234.

Because link-once template instantiation repeatedly compiles templates, link-once template instantiation may result in longer compile times, although the initial build of a project may be faster since prelink-time template instantiation will rebuild objects during the initial prelink phase. Link-once template instantiation is often used in build environments such as **clearmake**, where some aspect of prelink-time template instantiation is undesirable, although the prelinker may still need to instantiate export templates.

Instantiation Pragma Directives

Instantiation `#pragma` directives can be used to control the instantiation of specific template entities or sets of template entities. This is sometimes used as an alternative to Link-Once Template Instantiation in environments where minimizing the amount of prelinking is desirable. There are three instantiation `#pragma` directives:

- `#pragma instantiate fn` — Instructs the compiler to instantiate *fn*.
- `#pragma can_instantiate fn` — Instructs the compiler to consider *fn* for instantiation. This `#pragma` is often used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity turns out to be required.
- `#pragma do_not_instantiate fn` — Instructs the compiler to not instantiate *fn*. This `#pragma` is often used to suppress the instantiation of an entity for which a specific definition will be supplied.

Each of the above instantiation `#pragma` directives takes an argument, *fn*, which may be one of the following:

- A template class name (e.g., `A<int>`)

- A template class declaration (e.g., `class A<int>`)
- A member function name (e.g., `A<int>::f`)
- A static data member name (e.g., `A<int>::i`)
- A static data declaration (e.g., `int A<int>::i`)
- A member function declaration (e.g., `void A<int>::f(int, char)`)
- A template function declaration (e.g., `char* f(int, float)`)

A `#pragma` directive in which the argument is a template class name is equivalent to repeating the directive for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the `#pragma do_not_instantiate` directive. For example:

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the `#pragma instantiate` directive, and no template definition is available, or a specific definition is provided, an error is issued.

```
template <class T> void f1(T); // No body provided
template <class T> void g1(T); // No body provided
void f1(int) {} // Specific definition
void main()
{
    int i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}
#pragma instantiate void f1(int) // error - specific
                                // definition
#pragma instantiate void g1(int) // error - no body
                                // provided
```

f1(double) and **g1(double)** will not be instantiated (because no bodies were supplied) but no errors will be produced during the compilation (if no bodies are supplied at link time, a linker error will be produced).

A member function name (for example, `A<int>::f`) can only be used as a `#pragma` argument if it refers to a single user-defined member function (that is, a function that is not overloaded). Compiler-generated functions are not considered, so a name may refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration. For example:

```
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation directive may not be a compiler-generated function, an inline function, or a pure virtual function.

Implicit Inclusion

Implicit inclusion permits the compiler to assume that if it needs a definition to instantiate a template entity declared in a `.h` file it can implicitly include the corresponding `.C` file to get the source code for the definition. To enable this feature:



Set the **C/C++ Compiler**→**C++**→**Templates**→**Implicit Source File Inclusion** option to **On** (`--implicit_include`).

For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler searches for an appropriately suffixed C++ source file whose base name is `xyz`; if the file is found, the compiler processes it as if the file were included at the end of the main source file.

To find the template definition file for a given template entity, the compiler needs to know the full pathname of the file in which the template was declared and whether the file was included using the system include syntax (e.g., `#include <file.h>`). This information is not available for preprocessed source code containing `#line` directives. Consequently, the compiler will not attempt implicit inclusion for source code containing `#line` directives.

The following suffixes are searched for: `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, and `.cc`.

Care should be taken when writing files that may be implicitly included. For instance, if the file **xyz.C** is implicitly included since it contains a definition of a template declared in **xyz.h**, it is possible that more than one module will include it. In fact, most likely any module that includes **xyz.h** will end up implicitly including **xyz.C**. In addition, if **xyz.C** defines a global symbol (a non-template function or a variable), every module which (implicitly) includes it will get that symbol definition. This can lead to multiply defined symbol errors from the linker. This can be averted by having only template definitions in files that can be implicitly included. An implicitly included header file should be organized much the same way as any other header file.

Since a file is automatically pulled in when the implicit inclusion is on, the compiler driver or the builder do not need to separately compile it; just like a regular header file does not need to be separately compiled.

If a source file is unexpectedly compiled via implicit inclusion, you may see strange errors during compilation or linking. We recommend that you do not enable implicit inclusion when you start new projects written in standard C++.

Exported Templates

Exported templates are templates declared with the keyword `export`, which is not recognized by default. To enable this feature:



Set the **C/C++ Compiler→C++→Templates→Recognition of Exported Templates** option to **On (--export)**.



Note This option requires and will automatically enable **C/C++ Compiler→C++→Templates→Dependent Name Processing (--dep_name)**.

Exported templates and implicit inclusion (see “Implicit Inclusion” on page 640) are mutually exclusive. Enabling **C/C++ Compiler→C++→Templates→Recognition of Exported Templates (--export)** will automatically disable **C/C++ Compiler→C++→Templates→Implicit Source File Inclusion (--no_implicit_include)**

Exporting a class template is equivalent to exporting each of its static data members and each of its non-inline member functions. An exported template is special because its definition does not need to be present in a translation unit that uses that template. In other words, the definition of an exported (non-class) template does not need to be explicitly or implicitly included in a translation unit that instantiates that template. For example, the following is a valid C++ program consisting of two separate translation units:

```
// File 1:
#include <stdio.h>
static void trace() { printf("File 1\n"); }

export template <class T> T const & min(T const &, T const &);
int main() {
    trace();
    return min(2, 3);
}

// File 2:
#include <stdio.h>
static void trace() { printf("File 2\n"); }

export template <class T> T const & min(T const &a,
                                       T const &b) {
    trace();
    return a < b ? a : b;
}
```

Note that these two files are separate translation units: one is not included in the other. That allows the two functions `trace()` to coexist (with internal linkage).

Multiple and Virtual Inheritance

In situations involving multiple and/or virtual inheritance, the Green Hills C++ compiler has a limitation on the maximum byte offset of a base class within a derived class. If the offset is larger than the maximum value that fits in the unsigned short type, virtual function calls and RTTI might not work properly. For example:

```
class BIG_BASE {
public:
    char c[70000];
    virtual void foo();
};

class TWO_BASE {
public:
    int i;
    virtual void bar();
};

// offset of TWO_BASE is sizeof(BIG_BASE), which doesn't fit
// into 'unsigned short' on most architectures - diagnostic is given
class DERIVED : public BIG_BASE, public TWO_BASE {
public:
    // ...
};
```

In this type of situation, an error message is generated.

Where only single class inheritance is involved, the offset of a base class is always 0, so this limitation does not apply.

Exception Handling

When exception handling is enabled, memory is allocated from the heap to track the information needed to throw and catch exceptions. The amount of memory used depends upon the depth of the function call stack and the number of destructible objects. Two counters are provided to track the amount of memory in bytes reserved for exception handling from the heap. The amount of memory currently in use is tracked by the `__ghs_eh_mem_inuse`

variable. The maximum amount of memory ever in use is tracked in `__ghs_eh_mem_high`. Both counters are long long variables and can be accessed with extern declarations:

```
extern long long __ghs_eh_mem_inuse;  
extern long long __ghs_eh_mem_high;
```

Namespace Support

Namespaces are enabled by default. To disable this feature:



Set the **C/C++ Compiler→C++→Namespaces→Namespace Support** option to **Off** (`--no_namespaces`).

When looking up names in a template instantiation, some names must be found in the context of the template definition, while others can also be found in the context of the template instantiation.

The Green Hills C++ compiler implements two different instantiation lookup algorithms:

- The algorithm mandated by the standard (“Dependent Name Lookup” on page 645).
- The algorithm that existed before the dependent name lookup was implemented (“Lookup Using the Referencing Context” on page 645).

To enable dependent name lookup:



Set the **C/C++ Compiler→C++→Templates→Dependent Name Processing** option to **On** (`--dep_name`).

Dependent Name Lookup

When using dependent name lookup, the Green Hills C++ compiler implements the instantiation name lookup rules that are specified in the standard. This processing requires that non-class prototype instantiations are performed, which requires that the code be written using the **typename** and **template** keywords that are required by the standard.

Lookup Using the Referencing Context

When you are not using dependent name lookup, the compiler uses a name lookup algorithm that approximates the two-phase lookup rule of the standard.

This is done in such a way that is more compatible with existing code and legacy compilers.

When a name is looked up as part of a template instantiation but is not found in the local context of the instantiation, the name is looked up in a synthesized instantiation context that includes both names from the context of the template definition and names from the context of the instantiation.

For example:

```
namespace N {
    int g(int);
    int x = 0;
    template <class T> struct A {
        T f(T t) {return g(t);}
        T f() {return x;}
    };
}
namespace M {
    int x = 99;
    double g(double);
    N::A<int> ai;
    int i = ai.f(0);    // N::A<int>::f(int) calls
                        // N::g(int)
    int i2 = ai.f();    // N::A<int>::f() returns 0
                        //      (= N::x)

    N::A<double> ad;
    double d = ad.f(0); // N::A<double>::f(double)
                        // calls M::g(double)
    double d2 = ad.f(); // N::A<double>::f() also
                        // returns 0 (= N::x)
}
```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point at which the template was defined.
- Functions from the context in which the template was referenced are considered for all function calls in the template. Functions from the referencing context should only be visible for “dependent” functions calls.

Argument Dependent Lookup

By default, the Green Hills C++ compiler performs argument-dependent (Koenig) lookup, as specified by the standard. Functions made visible by using argument-dependent lookup overload with those made visible by normal lookup.

Setting the `--g++` option can affect argument-dependent lookup in some cases. For instance, normally Koenig lookup is suppressed when normal lookup of an unqualified name finds a block-scope function declaration that is not a using-declaration, but in GNU mode Koenig lookup will continue despite such a declaration.

```
namespace M {
    struct A {};
    void g(A, double);
    void h(A, double);
    A operator+(A, double);
}
void g(M::A, int);
M::A operator+(M::A, int);
void f() {
    void h(M::A, int);
    M::A a1;
    g(a1, 1.0); // calls M::g(M::A, double)

    a1 + 1.0;    // calls M::operator+(M::A, double)
    h(a1, 1.0); // calls h(M::A, int) unless --g++ is set,
                // in which case M::h(M::A, double)
                // is called
}
```

Precompiled Header Files

It is often desirable to avoid repeatedly recompiling a set of header files, especially when they introduce many lines of code and the primary source files that `#include` them are relatively small. The Green Hills C++ compiler provides a mechanism to save to disk a copy of the compiled headers together with a data structure recording the present state of the project. Then, when you recompile the same source file (or others with the same set of headers), the compiler checks that this *precompiled header file* (PCH) is still valid and, if so, uses it in place of the original headers. In the right circumstances, this can produce a dramatic improvement in compilation time. Note that PCH files can take up a lot of disk space. For more information, see “Performance Issues” on page 653.

Manual PCH Processing

To manually create a PCH file:



Set the option to **Create and Use a New PCH** and enter a filename in the **Value** field (`--create_pch file`).

To manually specify an existing PCH file for use:



Set the option to **Create and Use a New PCH** and enter a filename in the **Value** field (`--use_pch file`).

If the specified PCH file is invalid (i.e., if its prefix does not match the prefix for the current primary source file), a warning will be issued and it will not be used.

By default, PCH files are created and searched for in the directory containing the primary source file. To specify an alternate directory for writing and reading:



Specify the path in the option (`--pch_dir file`).

This option will not affect PCH files specified with an absolute pathname.

By default, PCH files cannot be shared by source files in different directories, due to the potential ambiguity of include paths. If the project does not include header files with the same name in different directories, this check may be disabled, allowing source files in different directories to share a PCH file:



Set the option PCH Require Matching Directory to Off (`--no_pch_match_directory`).

Automatic PCH Processing

To enable automatic PCH processing:



Set the option to **Automatically Use and/or Create a PCH** (`--pch`).

The compiler will search for a valid PCH file to read in, and if one is not found will create one for use on a subsequent compilation.

The PCH file contains a snapshot of all the code preceding the header stop point. The *header stop point* is typically the first token in the primary source file that does not belong to a preprocessing directive, but it can also be specified directly by `#pragma hdrstop` (see “Other Ways to Control PCHs” on page 653) if that comes first. For example:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

The header stop point is `int` (the first non-preprocessor token) and the PCH file will contain a snapshot reflecting the inclusion of **xxx.h** and **yyy.h**. If the first non-preprocessor token or the `#pragma hdrstop` appears within a `#if` block, the header stop point is the outermost enclosing `#if`. The following illustrates a more complicated example:

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
int i;
#endif
```

Here, the first token that does not belong to a preprocessing directive is again `int`, but the header stop point is the start of the `#if` block containing it. The PCH file will reflect the inclusion of `xxx.h` and, conditionally, the definition of `YYY_H` and inclusion of `yyy.h`; it will not contain the state produced by `#if TEST`.

A PCH file will be produced only if the header stop point and the preceding code (mainly, the header files themselves) meet certain requirements:

- The header stop point must appear at file scope; it may not be within an unclosed scope established by a header file. For example, a PCH file will not be created in this case.

```
// xxx.h
class A {
// xxx.C
#include "xxx.h"
int i; };
```

- The header stop point must not be inside a declaration started within a header file, nor (in C++) may it be part of a declaration list of a linkage specification. For example, in the following case the header stop point is `int`, but since it is not the start of a new declaration, no PCH file will be created:

```
// yyy.h
static
// yyy.C
#include "yyy.h"
int i;
```

- Similarly, the header stop point must not be inside a `#if` block or a `#define` started within a header file.
- The processing preceding the header stop must not have produced any errors. Note that warnings and other diagnostics will not be reproduced when the PCH file is reused.

- There must have been no references to the predefined macros `__DATE__` or `__TIME__`.
- There must have been no use of the `#line` preprocessing directive.
- `#pragma no_pch` (see section “Other Ways to Control PCHs” on page 653) must not have appeared.
- The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers.

When a precompiled header is produced, it contains, in addition to the snapshot of the compiler state, some information that can be checked to determine under what circumstances it can be reused. This includes:

- The compiler version, including the date and time the compiler was built.
- The current directory (i.e., the directory in which the compilation is occurring).
- The command line options.
- The initial sequence of preprocessing directives from the primary source file, including `#include` directives.
- The date and time of the header files specified in `#include` directives.

This information comprises the PCH “prefix.” The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether the latter is applicable to the current compilation.

As an illustration, consider two source files:

```
// a.C
#include "xxx.h"
// Start of code
// b.C
#include "xxx.h"
// Start of code
```

When **a.C** is compiled with automatic PCH processing enabled, a precompiled header file named **a.pch** is created. Then, when **b.C** is compiled (or when **a.C** is recompiled), the prefix section of **a.pch** is read in for comparison with the current source file. If the command line options are identical, if **xxx.h** has not been modified, and so forth, then, instead of opening **xxx.h** and processing it

line by line, the compiler reads in the rest of **a.pch** and thereby establishes the state for the rest of the compilation.

It may be that more than one PCH file is applicable to a given compilation. If so, the largest (i.e., the one representing the most preprocessing directives from the primary source file) is used. For instance, consider a primary source file that begins with:

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If there is one PCH file for **xxx.h** and a second for **xxx.h** and **yyy.h** together, the latter will be selected (assuming both are applicable to the current compilation). Moreover, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a precompiled header file is created, it takes the name of the primary source file, with the suffix replaced by **.pch**. Unless a (**--pch_dir**) is specified (see “Other Ways to Control PCHs” on page 653), it is created in the directory of the primary source file.

When a precompiled header file is created or used, a message such as the following is issued:

```
"test.C": creating precompiled header file "test.pch"
```

To suppress this message:



Disable the option (**--no_pch_messages**).

When is set to **Automatically Use and/or Create a PCH (--pch)**, the compiler will consider a precompiled header file obsolete and delete it under the following circumstances:

- If the precompiled header file is based on at least one out-of-date header file but is otherwise applicable for the current compilation.
- If the precompiled header file has the same base name as the source file being compiled (e.g., **xxx.pch** and **xxx.C**) but is not applicable for the current compilation (e.g., because of different command line options).

This handles some, but not all, common cases. Other PCH file clean-up must be performed by the programmer.

Other Ways to Control PCHs

There are several ways you can control and/or tune how precompiled headers are created and used.

- `#pragma hdrstop` may be inserted in the primary source file at a point prior to the first token that does not belong to a preprocessing directive. This enables you to specify where the set of header files subject to precompilation ends. For example:

```
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
#include "zzz.h"
```

Here the precompiled header file will include processing states for **xxx.h** and **yyy.h** but not **zzz.h**. This is useful if you decide that the information added by what follows the `#pragma hdrstop` does not justify the creation of another PCH file.

- `#pragma no_pch` may be used to suppress precompiled header processing for a given source file.

Performance Issues

The relative overhead incurred in writing out and reading back in a precompiled header file is quite small for reasonably large header files.

In general, the cost of writing out a precompiled header file is relatively low, even if the file ends up not being used. If it is used, it almost always produces a significant increase in compilation speed. The problem is that the precompiled header files can be quite large (from a minimum of about 250K bytes to several megabytes or more).

Thus, despite the faster recompilations, precompiled header processing is not likely to be justifiable for an arbitrary set of files with nonuniform initial sequences of preprocessing directives. Rather, the greatest benefit occurs when a number of source files can share the same PCH file. The more sharing, the less disk space is used. With sharing, the disadvantage of large precompiled

header files can be minimized, without giving up the advantage of a significant speedup in compilation times.

Consequently, to take full advantage of header file precompilation, try to reorder the `#include` sections of your source files and/or group `#include` directives within a commonly used header file.

Different environments and different projects will have different needs, but in general you will need to experiment and may need to make some minor changes to source code in order to make the best use of this feature.

Linkage

C++ accepts the `extern "language"` storage class specifier in order to achieve linkage between C++ and C. The full syntax is as follows:

```
extern "language" {  
    declarations  
}
```

or

```
extern "language" declaration;
```

where *language* may be C or C++.

Language	Effect on Resulting Code
C	C++ does not alter the procedure name as it usually would when confronted with an overloaded function.
C++	Uses C++ naming rules. Function names are always mangled according to C++ linkage specifications. This is the default linkage.

Note that the `extern "language"` directive only affects the external names of functions so that the compiler will apply the appropriate function naming rules. This directive does not modify the type or number of arguments of a function, or its return type. Normal C++ type checking rules are not altered by this directive.

For more information about using C++ with C, see Chapter 18, “Mixing Languages and Writing Portable Code”.

Post Processing in C++

In C or C++, *global objects* (or non-local static objects) are those objects which are declared outside of the scope of any function and are available throughout the entire program. C++ objects which are instances of a class type have mechanisms for automatic construction/initialization and destruction/cleanup through the use of *constructors* and *destructors*. Constructors are functions which are automatically called by the compiler when an object is created. Destructors are called when an object is deleted. This implies some implementation-specific behavior for global objects which may vary from C++ system to C++ system.

Global objects, such as those in libraries (e.g. `cin`, `cout`, and `cerr`) must be constructed and initialized for the entire program. This means that the constructor functions must be called as soon as the program begins. The compiler has no knowledge of external global objects contained in other modules or libraries except for an `extern` declaration. This is not enough information for the compiler to be able to properly ensure that these calls are performed. For targets that use the Green Hills linkers, the linker resolves the global constructor and destructor information. In other environments, the **cxnmunch** utility assumes this responsibility. The VxWorks environment is unique in that the module load/unload functions invoke the global constructors/destructors, or else the user executes them manually.

After an executable has been produced, the Green Hills C++ compiler driver calls an **nm** utility (such as **gnm**) to find all global symbols. The output of the **nm** utility is sent to the post-link program **cxnmunch**. **cxnmunch** searches for all global constructor and destructor calls and generates a C module that will execute these calls appropriately at program startup and exit. The driver then invokes the compiler and assembler to produce another object module. Following that, the linker is invoked to relink the new constructor/destructor object with the original object modules and libraries. This produces the fully processed C++ executable, which has all of the appropriate constructor and destructor calls.

The driver makes all of this processing completely transparent. However, if you choose not to use the driver provided by Green Hills, you are responsible for calling the post-link program after producing an executable; otherwise the program may not run correctly.

The C++ decode Utility

The **decode** utility is provided with Green Hills C++. Its syntax is as follows:

decode [-u]

This utility is a C++ name “demangler”. It reads the input from `stdin` and writes output to `stdout`. Anything that looks like mangled names in the input are demangled. Everything else is passed through unchanged. This makes the program suitable, for example, as a filter for the output of a linker; error messages in general are passed through unaltered, but mangled names are demangled. For example:

```
ld: Undefined symbol
      f__Fif
```

is changed to:

```
ld: Undefined symbol
      f(int, float)
```

The decode utility takes only a single option: **-u**. When used, **-u** specifies that external names have an added leading underscore.

C++ Implementation-Defined Features

- **1.3.2: Diagnostic message** — For a complete list of the C++ compiler diagnostic messages, see the *MULTI: C and C++ Compiler Error Messages* book (not available in print).
- **1.3.5: Implementation-defined behavior** — Documentation is provided in this section.
- **1.4: Implementation compliance** — The standard Green Hills distribution is built around a hosted implementation with the standard set of available libraries.
- **1.7: The C++ memory model** — The size of a byte is 8 bits.
- **1.9: Program execution** — The minimum requirements are all fulfilled.
- **2.1: Phases of translation** — Mapping of file characters to the basic source character set is performed via identity mapping. Non-empty sequences of white-space characters (other than newline characters) are retained

during preprocessing. Sources of translation units containing definitions of templates that are being instantiated are required to be available.

- **2.2: Character sets** — ASCII encoding is used for all characters.
- **2.8: Header names** — See “Instructing the Compiler to Search for Your Headers” on page 122.
- **2.13.2: Character literals** — The **C/C++ Compiler→Special Tokens→Host & Target Character Encoding** option enables recognition of multi-byte character Kanji sequences. When this option is disabled, or when unrecognized sequences are encountered, up to four chars are generated, initializing an `int` value, the least significant byte of which is initialized to the rightmost c-char of the multi-byte character literal. If there are less than four c-chars, the most significant bytes will be 0. Character literals outside of the implementation-defined range are truncated to the [0,255] range. Universal characters are truncated in the same way as `\xhhh` characters.
- **2.13.3: Floating literals** — Same as the C compiler (see “ANSI C Implementation-Defined Features” on page 605).
- **2.13.4: String literals** — Duplicate string literals are merged by default. Enable the **C/C++ Compiler→C/C++ Data Allocation→Uniquely Allocate All Strings** option to create a unique instance of every string.
- **3.6.1: Main function** — `main()` is required for stand-alone projects, and all operating systems except for VxWorks. The following additional `main()` definitions are supported:

- `int main(int argc, char *argv[], char *env[])`

(where `*env[]` is a null-terminated array of environment variables.)

- A return type of `void` is also permitted with a warning.

The linkage of `main()` is external.

- **3.6.2: Initializations of non-local objects** — Dynamic initialization of objects of namespace scope is performed by `_main()`, except for VxWorks projects which use the `_ctors` array. All static constructors within one compilation unit are executed in lexical order. Constructors in different compilation units are executed in link-line order.
- **3.9: Types** — For data type sizes and alignments, see “PowerPC Characteristics” on page 104.

- **3.9.1: Fundamental types** — `char` and `unsigned char` hold equivalent values by default. For more information, see the **C/C++ Compiler→Data Types→Signedness of Char Type** option. `float` and `double` are represented in the IEEE format.
- **3.9.2: Compound types** — Pointers are represented as plain memory addresses in the natural way for the target.
- **4.7: Integral conversions** — Unrepresentable values are truncated to a value that fits into the destination type using modulo arithmetic.
- **4.8: Floating-point conversions** — Same as the C compiler (see “ANSI C Implementation-Defined Features” on page 605).
- **4.9: Floating-integral conversions** — Same as the C compiler (see “ANSI C Implementation-Defined Features” on page 605).
- **5.2.8: Type identification** — `typeid` expression results are only of static type `std::type_info`. No derived classes are provided.
- **5.2.10: Reinterpret_cast** — Mapping remains the same. Mapping of pointers to addresses and vice versa uses `int` or larger type.
- **5.3.3: Sizeof** — For information about the sizes of fundamental types, see “PowerPC Characteristics” on page 104.
- **5.6: Multiplicative operators** — Remainder from the division of a negative and non-negative operand is the same sign as the first operand.
- **5.7: Additive operators** — `ptrdiff_t` is a typedef for `int` on 32-bit targets, or `long` on 64-bit targets.
- **5.8: Shift operators** — The result of `E1 >> E2` where `E1` has a signed type and a negative value is negative.
- **7.1.5.2: Simple type specifiers** — Type `char` is unsigned by default. Bitfields are unsigned by default.
- **7.2: Enumeration declarations** — Enumerations have a type of `int`, unless the **C/C++ Compiler→Data Types→Use Smallest Type Possible for Enum** option is enabled.
- **7.4: The asm declaration** — `asm` declarations are unmodified by the compiler. For more information, see Chapter 19, “Enhanced asm Macro Facility for C and C++”.
- **7.5: Linkage specifications** — Routines declared with `extern "C"` will not have name mangling performed on them. The default mode, `extern "C++"`, performs name mangling. No other language linkages

are supported. Mangling of function names, type names, and static data member names is performed by the C++ compiler.

- **8.5.3: References** — In the circumstances described, a temporary is generated, rvalue is copied into it, and the reference is bound to the temporary.
- **9.6: Bit-fields** — A bitfield is allocated at the current offset, unless it crosses an offset which is a multiple of the size of the base type, in which case it is padded up to the first multiple of the size of the base type. Plain bitfields are unsigned unless **C/C++ Compiler→Data Types→Signedness of Bitfields** is set to **Signed**.
- **12.2: Temporary objects** — Are generated for `f(x(2))`, but the result will be stored directly into `b`.
- **14: Templates** — Only C and C++ linkages are supported.
- **14.7.1: Implicit instantiation** — Recursive instantiations are limited to 64.
- **15.5.1: The terminate() function** — The stack is unwound before `terminate()` is called.
- **15.5.2: The unexpected() function** — Unless reset by user, will call `terminate()`.
- **16.1: Conditional inclusion** — Numeric values for character literals in `#if` and `#elif` controlling expressions is identical to `char` literals in regular expressions.
- **16.2: Source file inclusion** — See “Instructing the Compiler to Search for Your Headers” on page 122.
 - `#include <h-char-sequence>` — searches the system header directories. All identifiers in *h-char-sequence* will be resolved.
 - `#include "q-char-sequence"` — searches the current directory, and then all include directories. Sequences in *q-char-sequence* will be treated as a string literal, and will not be resolved.
 - Mapping between the sequence and the external source filename — is performed through the identity function.
 - `#include` preprocessor directives can be nested to arbitrary depth.
- **16.6: Pragma directive** — See “Pragma Directives” on page 679.

- **16.8: Predefined macro names** — For a list of the required preprocessor macros and their values, see “Macro Names Required by ANSI C and C++” on page 664.
- **17.4.4.5: Reentrancy** — All subroutines are reentrant provided that the target system has been configured with the appropriate lock routines in subdirectory `src/libsys`.
- **17.4.4.8: Restrictions on exception handling** — See the *Dinkum C++ Library Reference* (included with your distribution, at `install_dir/scxx/html/index.html`)
- **18.1: Types** — `NULL` is defined as `(0)`.
- **18.3: Start and termination** — `exit()` returns the value of argument status:
 - `EXIT_SUCCESS` — has a value of 0.
 - `EXIT_FAILURE` — has a value of 1.
- **18.4.2.1: Class `bad_alloc`** — `what()` returns an empty string.
- **18.5 to 18.6** — Various class definitions. See *Dinkum C++ Library Reference*.
- **20.1.5: Allocator requirements** — Match those in the standard. Certain operations on containers and algorithms will behave differently if allocator instances compare non-equal. Instances of `std::allocator` always compare equal.
- **21.1.3.1: struct `char_traits<char>`** — Types `streampos` and `streamoff` are typedefs for `fpos<mbstate_t>` and `long`, respectively. Type `wstreampos` is a typedef for `streampos`.
- **22.2: Standard locale categories** — Various type definitions. See *Dinkum C++ Library Reference*.
- **22.2.5.1.2: time_get virtual functions** — Two-digit year numbers are supported. Numbers in the range [0,68] represent years 2000 to 2068. Numbers in the range [69,136] represent years 1969 to 2036. See *Dinkum C++ Library Reference*.
- **23.2 to 23.3** — Various type and class definitions. See *Dinkum C++ Library Reference*.
- **26.2.8: complex transcendentals** — `pow(0, 0) == (1, 0)`

- **26.3: Numeric arrays** — Various type and class definitions. See *Dinkum C++ Library Reference*.
- **27.1.2: Positioning Type Limitations** — In Green Hills standard libraries, `traits::pos_type` and `traits::off_type` are typedefs for `streampos` and `streamoff` respectively.
- **27.4.1: Types** — `streamoff` is a typedef for `long`.
- **27.4.2.4: ios_base static members** — The synchronization flag is ignored. Streams are always synchronized.
- **27.4.4.3: basic_ios iostate flags functions** — On failure, the object to be thrown is constructed with a string argument that is the first applicable of `ios_base::badbit` set, `ios_base::failbit` set, or `ios_base::eofbit` set.
- **27.7.1.3: Overridden virtual functions** — `basic_streambuf::setbuf()` has no effect.
- **27.8.1.4: Overridden virtual functions** — `basic_filebuf::setbuf()` with non-zero arguments invokes the C library function `setvbuf` with the given arguments, scaled appropriately, and with a mode argument of `_IOFBF`.
- **B: Implementation quantities** — In order to guard against infinite recursion, the compiler places limits on the following quantities:
 - Nesting levels for `#include` files (10 recursive inclusions of the same file; no general limit)
 - Recursively nested template instantiations (64).

Associated diagnostics include 3 and 456.

- **C.1.9.16.8: Predefined names** — `__STDC__` is defined as 0.
- **C.2.2.3: Macro NULL** — Is defined as `(0)`.
- **D.6: Old iostreams members** — See 21.1.3.1 for `streampos` and `streamoff` typedefs.

Deprecated C++ Headers

The Green Hills C++ libraries provide all 51 of the headers specified by the *International Standard ISO/IEC 14882:1998*, as well as the common non-standard headers `<hash_map>`, `<hash_set>`, and `<slist>`.

The following non-standard header files are provided only for the support of legacy C++ code. These header files are deprecated, and may be removed in a future release:

```
<fstream.h>  
<iomanip.h>  
<iostream.h>  
<new.h>  
<rope>  
<stdiostream.h>  
<stl.h>  
<strstream.h>
```

Macros, Pragma Directives, and Intrinsic Functions

Chapter 16

This Chapter Contains:

- Predefined Macro Names
- Pragma Directives
- Intrinsic Functions

Predefined Macro Names

The preprocessor defines a number of macro names, depending on the compiler and language you use and the builder and driver options you specify. To write code that compiles differently depending on which macro names the preprocessor defines, use the `#ifdef` preprocessor directive.

The following sections contain lists that cover all of the preprocessor predefined macro names.

Macro Formats

The standards for C and C++ require that all compiler-predefined macros begin with one or two underscore characters. Prior to the ANSI standard, some C preprocessors defined macros without leading underscores (such as `ghs` or `unix`) that could conflict with macro names in your program.

Each Green Hills macro is defined with two leading underscores and two trailing underscores in all language modes, unless stated otherwise.

Macros marked with an asterisk (*) are also defined by default without the two trailing underscores.

When using C or C++ modes other than Strict ANSI C, you can instruct the preprocessor to generate versions of macros marked with * that do not have a leading or trailing underscore by enabling the **Advanced→Preprocessor Options→Definition of Unsafe Symbols (--unsafe_predefines)** option.

All predefined macros have the value 1, unless stated otherwise.

Macro Names Required by ANSI C and C++

The standards for C and C++ require the preprocessor to define certain macro names. These macro names and their values are shown in the table below.

`__cplusplus`

The value of this macro is:

- 199711L — in ANSI C++
- 1 — in other modes of C++ (EC++, EEC++, ARM, GNU)
- Undefined in C

`__DATE__`A string literal representing the date of the current compilation, which is 11 characters long, and in the form: *mmm dd yyyy*. For example: `Apr 01 2001``__FILE__`A string literal representing the name of the current source file (for example, `file.c`). For the base file, `__FILE__` includes the pathname. If the compiler reaches the file through an `#include` directive, `__FILE__` contains the pathname (if any) from the include path through which the file was found. If the include path is relative, it is appended to the pathname of the base file.`__LINE__`

An integer literal representing the line number in the current source file.

`__STDC__`

The value of this macro is:

- 0 — in C++ and ANSI C mode
- 1 — in Strict ANSI C mode and C99 mode
- Undefined in K&R C

`__STDC_VERSION__`

The value of this macro is:

- 199901L — in C99 mode
- 199409L — in ANSI C and GNU C
- Undefined in C++ and K&R C

`__STDC_HOSTED__`

The value of this macro is:

- 1 — in C++ and all modes of C other than K&R C
- Undefined in K&R C

`__TIME__`A string literal representing the local time of the current compilation, which is 8 characters long, and in the form: *hh:mm:ss*. For example: `"11:46:51"`

Language Identifier Macros

These macros identify the language used:

Primary Language Macros

<code>__LANGUAGE_ASM__</code>	*
Language is assembly.	
<code>__LANGUAGE_C__</code>	*
Language is C.	
<code>__LANGUAGE_CXX__</code>	*
Language is C++.	

C Language Macros

<code>__GNUC__</code>
GNU C or C++ mode (-gcc or --g++).
<code>__Japanese_Automotive_C__</code>
Japanese Automotive C (see“Japanese Automotive C Extensions” on page 587).
<code>__PROTOTYPES__</code>
All modes of C except K&R mode. All modes of C++.
<code>__STRICT_ANSI__</code>
Language is Strict ANSI C or Strict C99 (-ANSI or -C99). Undefined in C++.

C++ Language Macros

<code>__cplusplus</code>
Language is C++ (this is for backwards compatibility with some older C++ implementations). When writing new code, use <code>__cplusplus</code> .
<code>__embedded_cplusplus</code>
Language is Embedded or Extended Embedded C++.
<code>__EMBEDDED_CXX</code>
Language is Embedded C++ (for example, option --e).

<code>__EMBEDDED_CXX_HEADERS</code>
The Embedded C++ header files and libraries are used (for example, options <code>--el</code> or <code>--ele</code>).
<code>__EXTENDED_EMBEDDED_CXX</code>
Language is Extended Embedded C++ (ESTL) (for example, option <code>--ee</code>).
<code>__EXTENDED_EMBEDDED_CXX_HEADERS</code>
The Extended Embedded C++ header files and libraries are used (for example, options <code>--eel</code> or <code>--eele</code>).
<code>__STANDARD_CXX</code>
Language is Standard C++.
<code>__STANDARD_CXX_HEADERS</code>
The Standard C++ header files and libraries are used (for example, options <code>--stdl</code> or <code>--stdle</code>).

Green Hills Toolchain Identification Macros

<code>__EDG__</code>
C or C++ compilers.
<code>__ghs__</code> *
Any Green Hills Software compiler or preprocessor.
<code>__ghs_asm</code>
Indicates that the Green Hills assembler is in use.
<code>__ghs_c_int__</code>
Predefined <code>typedef</code> equivalent in size to <code>int</code> . When used to describe a function parameter, it requires that the parameter passed in is an integer constant.
<code>__GHS_REVISION_DATE</code>
The date of the compiler, represented as a string similar to <code>__DATE__</code> .
<code>__GHS_REVISION_VALUE</code>
Represents the date of the compiler build in the <code>time_t</code> format.
<code>__GHS_VERSION_NUMBER</code>
Represents the version of the toolchain as a three-digit decimal integer. For example, in MULTI 5.0.6, this macro would be defined as 506.

<code>__gnu_asm</code>	Indicates that the GNU assembler is in use.
<code>__unix_asm</code>	Indicates that the UNIX assembler is in use.

PowerPC-Specific Predefined Macro Names

The following table lists predefined macros specific to the PowerPC processor family. For a complete list of individual processor macros, see “PowerPC Processor Variants” on page 114.

<code>__ALTIVEC__</code>	PowerPC CPU variant, supports AltiVec extensions.	
<code>__EFP_APU__</code>	Defined for any PowerPC processor that includes the EFP Arithmetic Processing Unit, such as the PowerPC 8540.	*
<code>__MAC__</code>	Target supports the IBM PowerPC 4xx family DSP/MAC extensions.	*
<code>__PowerPC__</code> <code>__ppc__</code>	PowerPC processor family.	*
<code>__SPE__</code>	Defined for PowerPC processors that includes the SPE Arithmetic Processing Unit, such as PowerPC 8540, but not if the -noSPE option is specified (see “Target Options” on page 269).	*
<code>__VEC__</code>	For processors that support AltiVec, defined as <code>10205</code> as specified by the AltiVec Programming Interface Manual. This macro is not defined if AltiVec has been disabled.	
<code>pixel</code>	Defined as <code>__pixel</code> . AltiVec keyword predefinition.	

vector
Defined as <code>__vector</code> . AltiVec keyword predefinition.
<code>__vle__</code>
Defined for PowerPC processors that support VLE. For a list of processors that support VLE, see “PowerPC Processor Variants” on page 114.

Endianness Macros

One of these symbols is always defined to specify the endianness of the target processor.

<code>__BIG_ENDIAN__</code>
Big Endian byte order.
<code>__LITTLE_ENDIAN__</code>
Little Endian byte order.

Data Type Macros

These macros specify the size and signedness of certain data types.

Sizes

<code>__CHAR_BIT</code>
Defined as the size of <code>char</code> in bits.
<code>__FUNCPTR_BIT</code>
Defined as the size of a function pointer in bits.
<code>__INT_BIT</code>
Defined as the size of <code>int</code> in bits.
<code>__LONG_BIT</code>
Specifies the size of <code>long</code> in bits.
<code>__LLONG_BIT</code>
Specifies the size of <code>long long</code> in bits. Defined only if the <code>long long</code> type is supported.

<code>__PTR_BIT</code>
Specifies the size of a pointer in bits.
<code>__REG_BIT</code>
Specifies the size of an integer register in bits.
<code>__SHRT_BIT</code>
Specifies the size of <code>short</code> in bits.
<code>__WCHAR_BIT</code>
Specifies the size of <code>wchar_t</code> in bits.

Signedness

<code>__Enum_Field_Is_Signed__</code> <code>__Enum_Field_Is_Unsigned__</code>
Bitfield enumerations are signed or unsigned.
<code>__Field_Is_Signed__</code> <code>__Field_Is_Unsigned__</code>
Bitfields are signed or unsigned.
<code>__Ptr_Is_Signed__</code> <code>__Ptr_Is_Unsigned__</code>
Pointers are signed or unsigned.
<code>__Char_Is_Signed__</code> <code>__SIGNED_CHARS__</code> <code>__Char_Is_Unsigned__</code> <code>__CHAR_UNSIGNED__</code>
Type <code>char</code> is signed or unsigned.
<code>__WChar_Is_Signed__</code> <code>__WChar_Is_Unsigned__</code>
Type <code>wchar_t</code> is signed or unsigned.

Floating-Point Macros

These macros specify the manner in which floating-point operations are performed.

<code>__DOUBLE_HL__</code>	*
Passed if the high word of a double comes at the lower address in memory. This is usually the case for big endian targets, although certain targets behave differently.	
<code>__IeeeFloat__</code>	*
IEEE-754 floating-point format. This symbol is always defined.	

Floating-Point Mode Macros

One of these symbols is always defined, unless full hardware floating point is enabled.

<code>__ghs_fprecise__</code>	*
Extra run-time code is produced to perform floating point operations that truncate exactly to a specified less precise type on a target that most efficiently operates on a more precise type.	
<code>__NoFloat__</code>	*
No floating point mode.	
<code>__SoftwareDouble__</code>	*
Double precision floating point uses integer instructions.	
<code>__SoftwareFloat__</code>	*
All floating point is done with integer instructions.	

MISRA Macros

These macros relate to the MISRA C variant.

<code>__MISRA_i</code>
MISRA diagnostic levels. <i>i</i> is either 8 or 118–127. The macro is defined as one of the following: • 0: Silent • 1: Warn • 2: Error For more information about the MISRA rules, see “MISRA C 1998” on page 216.
<code>__ONLY_STANDARD_KEYWORDS_IN_C</code>
MISRA rule No. 1

Memory Model Macros

These macros relate to various different forms of memory model.

Small Data Area Macros

<code>__ghs_sda</code>	*
Defined if Small Data Area optimization is supported. If this macro is not defined, instances of <code>#pragma ghs start_sda</code> will be ignored with a warning.	
<code>__ghs_sda_threshold</code>	*
Defined as the size, <i>n</i> , of the SDA threshold (as specified by the <code>-sda=<i>n</i></code> option).	

Zero Data Area Macros

<code>__ghs_zda</code>	*
Defined if Zero Data Area optimization is supported. If this macro is not defined, instances of <code>#pragma ghs start_zda</code> will be ignored with a warning.	
<code>__ghs_zda_threshold</code>	*
Defined as the size, <i>n</i> , of the ZDA threshold (as specified by the <code>-zda=<i>n</i></code> driver option).	

Global Register Macros

<code>__GlobalRegisters</code>
Corresponds to the <code>-globalreg=<i>n</i></code> option, where <i>n</i> is a non-negative positive integer.

Global Variable Macros

<code>__GHS_NOCOMMONS__</code>	*
Uninitialized global variables in C do not exhibit <code>COMMON</code> variable behavior (corresponds to the <code>--no_commons</code> driver option).	

Alignment and Packing Macros

These macros specify the size of data alignment and packing.

<code>__ghs_alignment</code>
Alignment of the most-aligned type, which is either <code>double</code> , <code>long</code> , or <code>long long</code> . Has a value of 1, 2, 4, or 8.
<code>__ghs_max_pack_value</code>
Represents the largest value that may be passed to <code>#pragma pack(n)</code> .
<code>__ghs_packing</code>
Represents value specified with the -pack=n option on the command line. Has a value of 1, 2, 4, or 8. If the macro name is not defined, the -pack= option was not specified on the command line. This value is not changed by the <code>#pragma pack(n)</code> directive.

Current File and Function Macros

These macros identify the current file and function. See also `__FILE__` in “Macro Names Required by ANSI C and C++” on page 664.

<code>__BASE__</code>
A string literal representing the name of the current source file, without the directory path.
<code>__FULL_DIR__</code>
A string literal representing the full pathname of the directory of the current source file.
<code>__FULL_FILE__</code>
A string literal representing the full pathname of the current source file.
<code>__FUNCTION__</code>
A string literal representing the current function name. The compiler does not resolve this macro when you use the -E or -P options.
<code>__PRETTY_FUNCTION__</code>
In C++, a string literal representing the fully qualified function name (for example, <code>mynamespace::myclass::method</code>). Note that overloaded functions with different signatures might have the same fully qualified name. In C, <code>__PRETTY_FUNCTION__</code> behaves the same as <code>__FUNCTION__</code> . The compiler does not resolve this macro when you use the -E or -P options.

Object Format Macros

These macros specify the object format.

<code>__COFF__</code>	*
Object file type is COFF.	
<code>__ELF__</code>	*
Object file type is ELF.	

Optimization Predefined Macro Names

These macros relate to the optimization strategy employed.

<code>__GHS_Inline_Memory_Functions</code>
Inlining of C Memory Functions (see “Inlining of C Memory Functions” on page 325).
<code>__GHS_Inline_String_Functions</code>
Inlining of C String Functions is enabled (see “Inlining of C String Functions” on page 326).
<code>__GHS_Optimize_Inline</code>
Automatic inlining is enabled (see “Automatic Inlining” on page 317).

C++ Macros

These macros relate to various features of C++.

<code>__ARRAY_OPERATORS</code>
Defined when array new and delete are enabled (that is, <code>operator new[]</code> and <code>operator delete[]</code>).
<code>_BOOL</code>
Defined when <code>bool</code> is recognized as a basic type (for example, as with the option --bool).
<code>__EDG_IMPLICIT_USING_STD</code>
Defined when the standard namespace <code>std</code> is implicitly used (as with the option --using_std).
<code>__EDG_RUNTIME_USES_NAMESPACES</code>
Indicates that the C++ Run-time libraries use namespaces.

__EXCEPTION_HANDLING
__EXCEPTIONS
Indicates C++ compiler is running in a mode that allows exception handling.
__NAMESPACES
Indicates C++ namespaces are accepted.
__RTTI
Indicates Run-Time Type Identification code accepted.
_WCHAR_T
Defined if <code>wchar_t</code> is a keyword.

Operating System-Specific Macros

These macros identify the target operating system.

INTEGRITY Macros

__INTEGRITY
INTEGRITY real-time operating system.

ThreadX Macros

__THREADX
ThreadX real-time operating system.
TX_ENABLE_EVENT_LOGGING
ThreadX event logging enabled.

Deprecated Macros

These macros are deprecated and may not be supported in future versions of MULTI.

<code>__Char_Is_Signed__</code>	
Type <code>char</code> is signed <code>char</code> .	
<code>__Char_Is_Unsigned</code>	
Type <code>char</code> is unsigned <code>char</code> .	
<code>__Int_Is_32</code>	
Type <code>int</code> is 4 bytes.	
<code>__Int_Is_64</code>	
Type <code>int</code> is 8 bytes. (The macro <code>__Int_Is_64</code> is an old style macro, kept for backward compatibility. The new style macros (like <code>__INT_BIT=n</code>) should be used when possible.)	
<code>__msw</code>	*
Any version of Microsoft Windows (deprecated).	
<code>__LL_BIT</code>	
Describes the size of <code>long long</code> in bits.	
<code>__LL_Is_64</code>	
Type <code>long long</code> is 8 bytes (when <code>long long</code> is an allowed type).	
<code>__Long_Is_32</code>	
Type <code>long</code> is 4 bytes.	
<code>__Long_Is_64</code>	
Type <code>long</code> is 8 bytes.	
<code>__Ptr_Is_32</code>	
Pointers are 4 bytes.	
<code>__Ptr_Is_64</code>	
Pointers are 8 bytes.	
<code>__Reg_Is_32</code>	
CPU has 32-bit registers.	
<code>__Reg_Is_64</code>	
CPU has 64-bit registers.	
<code>__WChar_Is_Int__</code>	
Type <code>wchar_t</code> are <code>int</code> or unsigned <code>int</code> .	

<code>__WChar_Is_Long__</code>
Type <code>wchar_t</code> are long or unsigned long.
<code>__WChar_Is_Short__</code>
Type <code>wchar_t</code> is short or unsigned short.

Pragma Directives

#pragma directives allow individual compiler implementations to add special features to C programs without changing the C language. Programs that use #pragma directives stay relatively portable, although they make use of features not available in all ANSI C implementations.

According to the ANSI standard, the #pragma directives that are not recognized by the compiler should be ignored. This requirement may help when porting code between compilers because one compiler will recognize and process a certain #pragma directive, while a different compiler (which does not need or recognize that #pragma directive) will, by default, generate a warning and ignore it.

If a #pragma directive which would ordinarily be recognized is misspelled, the compiler will not recognize it and will, by default, generate a warning and ignore it. You can increase the severity of these messages to errors, or suppress them, as follows:



Set the **Compiler Diagnostics→C/C++ Messages→Unknown Pragma Directives** option to **Errors** (`--unknown_pragma_errors`) or **Silent** (`--unknown_pragma_silent`).

The majority of Green Hills proprietary #pragma directives begin with the keyword `ghs` to differentiate them from other implementations. The compiler considers any #pragma beginning with the `ghs` keyword to be recognized.

If the compiler recognizes a #pragma directive, it will perform some limited syntax checking. #pragma directives that fail these checks will, by default, generate warnings, and be ignored. You can increase the severity of these messages to errors, or suppress them, as follows:



Set the **Compiler Diagnostics→C/C++ Messages→Incorrect Pragma Directives** option to **Errors** (`--incorrect_pragma_errors`) or **Silent** (`--incorrect_pragma_silent`).

General Pragma Directives

The general #pragma directives are listed in the following table:

<code>#pragma alignfunc (n)</code>
Instructs the compiler to make the next function <i>n</i> -byte aligned, where <i>n</i> is 1, 2, 4, 8, 16, 32, 64, or 128.
<code>#pragma alignvar (n)</code>
Instructs the compiler to make the next variable <i>n</i> -byte aligned, where <i>n</i> is a power of 2 and no greater than 4096. If the next variable is a non-static local variable, this directive will cause the stack to be re-aligned at run time to align the variable.
<code>#pragma asm</code> <code>#pragma endasm</code>
These directives mark the beginning and end of a section of code which is to be copied literally to the assembly language output file. Any macros or preprocessing directives will be processed, but no processing of comments will be performed. The construct <code>#pragma endasm</code> may not contain any comments or \ escapes, although it may contain spaces or tabs before and after the # character, as long as the total length of the line does not exceed 255 characters. Note that this directive is not supported with C++ source code.
<code>#pragma ident "string"</code>
Inserts <i>string</i> in the output assembly file as a comment.
<code>#pragma inline function-list</code>
This #pragma is deprecated. Instructs the compiler to consider each function in the comma-separated <i>function-list</i> as a candidate for inlining. Note that you cannot force the compiler to inline any function.
<code>#pragma instantiate fn</code> <code>#pragma can_instantiate fn</code> <code>#pragma do_not_instantiate fn</code>
These directives can be used to control the instantiation of a template function, <i>fn</i> . For full documentation, see “Template Instantiation” on page 634.
<code>#pragma intvect intfunc integer_constant</code>
Instructs the compiler to insert a jump to the function <i>intfunc</i> at the address specified by <i>integer_constant</i> . For more information, see “Writing Interrupt Routines” on page 147.

```
#pragma message "string" #pragma message("string")
```

Prints *string* to the error output while compiling the file.

```
#pragma once
```

If this `#pragma` is placed at the beginning of a header file, the compiler will only include the file once, and will ignore subsequent `#include` statements that reference the file.

```
#pragma pack (n)
```

```
#pragma pack ()
```

```
#pragma pack (push {, name} {, n})
```

```
#pragma pack (pop {, name} {, n})
```

Instructs the compiler to lay out subsequent `class`, `struct`, and `union` definitions in memory with a maximum alignment of *n*, where *n* is a power of two and less than or equal to the value of `__ghs_max_pack_value`.

The second format, where *n* is not specified, reverts the maximum alignment to the default value, or that specified by the Builder or driver.

The third and fourth formats allow you to push and pop maximum alignment values, and to specify a *name* for the pushed value, so that it can subsequently be popped back to by name.

```
#pragma __printf_args
```

```
#pragma __scanf_args
```

These directives should be used immediately before a function declaration or definition.

If the function has a variable number of arguments and the last specified argument is of type `char *`, the function is treated as a variadic function like `printf` or `scanf`.

The compiler will expect that the last specified argument is a character string, with **printf** or **scanf** formatting sequences (that is "%s"), and will check the extra arguments to ensure that the type and number match the type and number specified in the format string.

```
#pragma unknown_control_flow (function-list)
```

Instructs the compiler to anticipate an *unexpected flow* from each function in the comma-separated *function-list*. Such a function would be `setjmp`. This information is used in various optimizations.

```
#pragma weak foo
```

Defines *foo* as a *weak symbol*. This `#pragma` may be used in either of the following ways:

- If *foo* is declared as an external reference in *filename*:

```
#pragma weak foo
extern int foo;
```

and *foo* is not defined in any other file, then the address of *foo* is set to zero. This will prevent an undefined symbol linker error.

- If *foo* is defined in *filename*:

```
#pragma weak foo
int foo = 1;
```

and a definition of *foo* appears in any other file, then that other definition will have priority, and the definition in *filename* will be treated as if it were an external reference. If all definitions are weak and there is more than one definition, it is undefined which definition is used.

By default, the linker does not pull in object files from libraries to resolve weak symbols. To force the linker to do so, use the **-extractweak** linker option.

```
#pragma weak foo = bar
```

This `#pragma` has the same effect as `#pragma weak foo`, except that (subject to the same conditions) the address of *foo* is set to the address of *bar*. It requires that *foo* is not defined in the current module, although it could be an external reference. *bar* must be defined at the outermost level of the current module. *bar* may not be a common symbol (see the **C/C++ Compiler→C/C++ Data Allocation→Allocation of Uninitialized Global Variables** option in “C/C++ Data Allocation” on page 227). This `#pragma` may be used to provide a backwards-compatible alias to a symbol which has been renamed in a new version of a program, as follows:

```
#pragma weak oldname=name
int name()
{
    return 5;
}
```

If any old software depends on the existence of *oldname*, it will continue to work as if the function is called *oldname*. If there are no longer any references to *oldname*, then the symbol effectively disappears and can be reused.

When aliasing variables, behavior is undefined if you use more than one way to reference the actual variable.

Green Hills Extension Pragma Directives

The Green Hills extension #pragma directives are listed in the following table:

```
#pragma ghs alias newsym oldsym
```

Creates a new global symbol *newsym* with the same address as *oldsym*. *newsym* cannot be defined in the current module or the behavior is undefined. *oldsym* must be defined at the outermost level of the current module. *oldsym* cannot be a common symbol (see the **C/C++ Compiler→C/C++ Data Allocation→Allocation of Uninitialized Global Variables** option in “C/C++ Data Allocation” on page 227).

`#pragma ghs alias newsym oldsym` is similar to `#pragma weak newsym=oldsym` except that in the latter case, *newsym* is defined as a weak symbol.

When aliasing variables, behavior is undefined if you use more than one way to reference the actual variable.

```
#pragma ghs check=(check-list)
```

Controls various run-time checks. Any run-time checks enabled either at the point of a function declaration (the prototype or the first unprototyped use) or at the point of a function definition will cause those run-time checks to be enabled for that function. Use the following words in the *check-list*.

Memory Checks. These checks do not affect other checks:

- `defaultmemory` — **Revert** to the `check=memory` or `check=nomemory` option as specified by the Builder or driver (see below).
- `memory` — **Memory Allocation (Intensive)** check

Other Checks. These checks do not affect memory checks:

- `all` — **All** checks.
- `assignbound` — **Assignment Bounds** check
- `bounds` — **Array Bounds** check
- `default` — **Revert** to the checks specified by the Builder or driver (see below).
- `nilderef` — **Nil Pointer Dereference** check
- `none` — **No** checks.
- `return` — **Return Without Value** check
- `switch` — **Case Label Bounds** check
- `watch` — **Write to Watchpoint** check
- `zerodivide` — **Divide by Zero** check

To turn off any of these checks, prepend `no` to the word. For example, to turn on all checks except **Divide by Zero**, use the following directive:

```
#pragma ghs check=(all,nozerodivide)
```

These directives require that you link in the Green Hills library `libind.a`.

For information about using run-time checks, see the **Debugging→Run-Time Error Checks** option in “Debugging Options” on page 187, and Chapter 14, “Viewing Memory Allocation Information” in the *MULTI: Debugging* book.

```
#pragma ghs includeonce
```

This directive is deprecated and may be removed in future versions of **MULTI**.

Instructs the compiler to include header files only once. If the compiler encounters subsequent `#include` directives relating to the same file, it will ignore them.

The same behavior can be obtained with the **-includeonce** driver option (see “**Process #include Directives**” on page 299).

```
#pragma ghs inlineprologue
```

```
#pragma ghs noinlineprologue
```

These directives control the inlining of function prologue and epilogue code.

`#pragma ghs inlineprologue` tells the compiler to inline the prologue and epilogue methods. This is useful when the prologue or epilogue routines are not callable.

`#pragma ghs noinlineprologue` lifts this restriction from the compiler and selects the most efficient method, given the optimization settings and the registers that must be saved.

This behavior can also be controlled with the **Advanced→Target Options→Function Prologues** Builder option and associated driver option (see “Target Options” on page 269).

```
#pragma ghs interrupt
```

Instructs the compiler to make the encompassing function, or the next function if in the file scope, an interrupt handling function.

The optional parameter `nonreentrant` instructs the compiler to assume the interrupt handler cannot be interrupted, resulting in a smaller and faster interrupt handler.

```
#pragma ghs nofloat interrupt
```

Instructs the compiler to prevent the encompassing function, or next function if in file scope (which must be separately identified as an interrupt function) from saving and restoring floating point-registers.

```
#pragma ghs nowarning n
#pragma ghs nowarning all
#pragma ghs endnowarning
```

Temporarily suppress diagnostic number *n* or all warnings. You may use multiple and/or nested `#pragma ghs nowarning` directives, but each should have a matching `#pragma ghs endnowarning`.

To display error and warning numbers with diagnostic messages, enable the **Compiler Diagnostics→C/C++ Messages→Varying Message Severity→Display Error Message Numbers** option (`--display_error_number`).

Note: The location where `#pragma ghs nowarning` must be is dependent on the implementation of the warning in question, and is not always at the position where the warning is shown in the diagnostic output.

A good general strategy for figuring out what lines you need to surround with these directives in order to suppress a warning is as follows. Start off by surrounding the line that gives the warning, then move on to surrounding the block if that does not suppress the warning, and then move on to surrounding the function if that does not suppress the warning. In a few cases you may need to leave off the `#pragma ghs endnowarning` entirely, so that the warning is disabled at the end of the file.

For example, warning #550 (variable was set but never used) can be disabled for a block-scope variable declaration by surrounding the block or function (not the line), but for static file-scope variable declarations, the `#pragma ghs endnowarning` must be left off entirely, and it only possible to disable warning #550 for all file-scope variables in the file or for none of them.

```
#pragma ghs Ostring
#pragma ghs ZO
```

`#pragma ghs Ostring` turns on optimizations, while `#pragma ghs ZO` turns them off. The optional *string* can contain any or all of the following letters:

- L — Loop optimizations
- M — Memory optimizations
- S — Small (but Slow) optimizations
- V — Vectorizing optimizations

Note: You can enable optimizations by omitting *string* from the directive.

For information about Builder and driver optimization options, see “Optimization Options” on page 176. For descriptions of many of our optimizations, see Chapter 7, “Optimizing Your Programs”.

```
#pragma ghs reference sym
```

Creates a reference to symbol *sym*, which can force the symbol to be pulled in from a library. It can also be used to prevent an unused function called *sym* from being deleted by the “Deletion of Unused Functions” optimization.

```
#pragma ghs revertoptions
```

Disables all options set via `#pragma ghs` directives and returns to the defaults specified by the Builder or driver.

```
#pragma ghs safeasm
```

```
#pragma ghs optasm
```

These directives mark the next hand-written assembly language entity (for example, `#pragma startasm`, `asm` macro, etc.) as “safe” for linker optimization analysis. Normally, hand-written assembly entities disqualify a module for linker optimization. If the entity is preceded by either of these directives, that entity does not disqualify the module for linker optimization. In order for a module to be considered “safe”, each assembly entity must be marked with one of these pragma directives.

`#pragma ghs optasm` marks the entity as safe and allows optimizations to occur within the entity.

`#pragma ghs safeasm` marks the entity as safe for some optimizations, but unsafe for others. If you want the final executable to contain the exact assembly code you have written, do not use this `#pragma` directive.

Only use these directives to mark assembly that does not contain any “unsafe” code. Unsafe code includes, but is not limited to, branches and control flow structures not normally generated by the compiler, and any code that violates the standard calling convention.

```
#pragma ghs section secttype="sectname"
```

Instructs the compiler to allocate to *sectname* code or data that would normally be allocated to *secttype*. *sectname* must not contain spaces, quotes, parentheses, or wildcards. For more information, see “Custom Program Sections” on page 128.

```
#pragma ghs startdata-type
```

```
#pragma ghs enddata-type
```

These directives mark the beginning and end of a section of code in which all data should be allocated to a particular type of data section, where *data-type* is one of the following:

- *data* — normal data sections.
- *sda* — *small data area* sections (see “Special Data Area Optimizations” on page 137).
- *zda* — *zero data area* sections (see “Special Data Area Optimizations” on page 137).

These directives will force the compiler to re-allocate data items to special data areas even if the **Target→Text and Data Placement→Special Data Area** option has not been enabled. The directives will also override any threshold which has been set with these options in order to control the maximum size of data items which can be allocated to a special data area.

You cannot nest these directives; each `#pragma ghs startdata-type` must be completed by a matching `#pragma ghs enddata-type` before another can be used.

```
#pragma ghs start_externally_visible
```

```
#pragma ghs end_externally_visible
```

These directives mark the beginning and end of a section of code where all symbols should be treated by optimizations as externally visible. Any functions or variables defined in this section will be considered visible outside of the source during Wholeprogram Optimizations. You can similarly specify that symbols are externally visible using the **-external=** and **-external_file=** options, or by using the `externally_visible` GNU attribute. For more information about the use of this `#pragma`, see “Wholeprogram Optimizations” on page 333.

```
#pragma ghs startnoinline
```

```
#pragma ghs endnoinline
```

These directives mark the beginning and end of a section of code such that any function defined in this section will not be considered for inlining. Even inlined C++ member functions or functions marked with the keywords `__inline` or `inline` will not be inlined. Template functions and member functions of template classes are not affected.

For more information about controlling inlining, see “Inlining Optimizations” on page 317.

```
#pragma ghs startnomisra
```

```
#pragma ghs endnomisra
```

These directives mark the beginning and end of a section of code in which the compiler does not perform MISRA checking.

For more information, see “MISRA C 2004” on page 197.

```
#pragma ghs static_call routine
```

Instructs the compiler to record a static call from the encompassing function to *routine* in the debug information if static calls are being recorded. If in file scope, the next function definition or declaration is used instead of the encompassing function, and that function must be defined at some point in the current module. For example, this directive helps support debug information when you have calls through function pointers.

```
#pragma ghs struct_min_alignment(n)
```

```
#pragma ghs struct_min_alignment()
```

Instructs the compiler to lay out subsequent `class`, `struct`, and `union` definitions in memory with a minimum alignment of *n* bytes.

The second format, where *n* is not specified, reverts the minimum alignment to the default value of 1.

The `#pragma` should not be active before including headers, as the structures referenced by the header need to be consistent between the module including the header and any other modules on which the header depends.

The **C/C++ Compiler**→**Alignment & Packing**→**Packing (Maximum Structure Alignment)** (`-pack=n`) and `#pragma pack(n)` directive override this pragma directive.

Conditional Pragma Directives

This feature is deprecated and may be removed in future versions of MULTI.

Any of the Green Hills extension `#pragma` directives can be made conditional upon whether you are generating debugging information or are optimizing your executable. To make a directive conditional, use the syntax:

```
#pragma ghs [condition] pragma-name
```

where the optional *condition* parameter is one of:

- `ifdebug` — Enable the `#pragma` only if debugging information is being generated (see **Debugging**→**Debugging Level** in “Debugging Options” on page 187).
- `ifnodebug` — Enable the `#pragma` only if debugging information is not being generated.
- `ifoptimize` — Enable the `#pragma` only if optimizations are enabled (see **Optimization**→**Optimization Strategy** in “Optimization Options” on page 176).
- `ifnooptimize` — Enable the `#pragma` only if optimizations are not enabled.

For example, to enable the **Assignment Bounds** and **Unallocated Memory** run-time checks only if optimizations are not enabled, enter `#pragma ghs check= (check-list)` as follows:

```
#pragma ghs ifnooptimize check=(assign,memory)
```

The `_Pragma` preprocessing operator from C99 can be used in all language modes. It is useful in cases when the effect of a directive is desired within a macro.

For example, the following can be used to define a macro DECL_ALIGN_BUF:

```
#define PRAGMA(x) _Pragma(#x)
#define ALIGNVAR(aligned) PRAGMA( alignvar(aligned) )
#define CHANGE_SEC(sec, name) PRAGMA( ghs section sec = name )
/* Define a buffer in section "sect" with the given name, size, and
 * alignment
 */
#define DECL_ALIGN_BUF(name, size, align, sect) \
ALIGNVAR(aligned) \
CHANGE_SEC(bss, sect) \
char name[size]; \
CHANGE_SEC(bss, default)
```

The macro can be invoked as:

```
DECL_ALIGN_BUF(mybuf, 80, 8, ".mybss")
```

This macro invocation has the effect of:

```
#pragma alignvar(8)
#pragma ghs section bss=".mybss"
char mybuf[80];
#pragma ghs section bss=default
```

Intrinsic Functions

Intrinsic functions perform certain tasks which are difficult or inefficient to write in a high-level language.

The name of a C or C++ intrinsic function begins with two underscores (__). It usually generates optimized inline code, often using special instructions that do not correspond to standard C and C++ operations.

Prototypes for the functions can be found in *install_dir/include/ppc/ppc_ghs.h*, and can be included as follows:

```
#include <ppc_ghs.h>
```

```
unsigned int __LWARX(unsigned int *p, unsigned int i);  
unsigned int __STWCX(unsigned int *p, unsigned int i,  
unsigned int v);
```

__LWARX returns p[i], and creates a reservation for use by __STWCX.

__STWCX returns 1 if v was stored to p[i], or 0 if the reservation was lost.

These intrinsics can be used to perform atomic memory accesses. For example the following code atomically sets the variable x:

```
int x;  
while (!__STWCX(&x, 0, __LWARX(&x, 0) | 1));
```

```
unsigned int __MFTB(void);  
__MFTBU()
```

The __MFTB() function returns the value of the time base register TB. The __MFTBU() function returns the value of the time base register TBU. The mftb instruction is used to read the time base registers when available otherwise the mfspr instruction is used.

```
void *__builtin_return_address(unsigned int level);
```

Returns the return-address of the current function. The *level* argument must be set to 0. This function can be used in both normal and interrupt procedures.

```
void __builtin_set_return_address(unsigned int level, void
*addr);
```

Sets the current return address to *addr*. The *level* argument must be set to 0. This function can be used in both normal and interrupt procedures.

This function can be used together with `__builtin_return_address()` to adjust the return address of an interrupt procedure in order either to re-execute or to avoid executing an offending instruction that may have caused the interrupt. It should be used with caution elsewhere as it may lead to unintended program execution.

```
void __DI();
void __EI();
```

These functions disable and enable interrupts and can bracket a critical section of code that must not be interrupted.

These functions clear and set `MSR[EE]` (bit 16 of the MSR, where bit 0 is the most significant bit.) The `__DI` function clears the bit and the `__EI` function sets the bit.

The compiler does not output any synchronization instructions before or after these intrinsics. Depending on the state of the processor and caches, you may need to insert synchronization instructions manually.

```
void __EIEIO(void);
void __eieio(void);
void __ISYNC(void);
void __isync(void);
void __LWSYNC(void);

void __MBAR(void);

void __MSYNC(void);

void __PTESYNC(void);
void __SYNC(void);
void __sync(void);
void __TLBSYNC(void);

void __WAIT(void);
```

These functions insert synchronization instructions. The instruction inserted has the same name as the intrinsic function.

```
void __MTDCR(unsigned int dcr, unsigned int val);
unsigned int __MFDCR(unsigned int dcr);
```

The `__MTDCR()` function sets the device-control-register number *dcr* to the value *val*. The `__MFDCR()` function returns the value in the device-control-register number *dcr*. For both functions, *dcr* must be a constant in the range 0-1023.

```
void __MTSPR(unsigned int spr, unsigned int val);  
unsigned int __MFSPR(unsigned int spr);
```

The `__MTSPR()` function sets Special-Purpose-Register number `spr` to the value `val`.

The `__MFSPR()` function returns the value in Special-Purpose-Register number `spr`. For both functions, `spr` must be a constant in the range 0-1023.

```
void __RIR(unsigned int);
```

Takes an `unsigned int` key previously returned by `__DIR()` or `__EIR()` and restores interrupts to the state represented by that key.

This function, together with `__DIR()` and `__EIR()` can be used to safely enable or disable interrupts and later restore them to their original state when that state is not known at compile time.

The compiler does not output any synchronization instructions before or after this intrinsic. Depending on the state of the processor and caches, you may need to insert synchronization instructions manually.

```
double __MFFS(void);  
double __mffs(void);  
void __MTFSF(double frb);  
void __mtfsf(double frb);  
double __setflm(double frb);
```

The `__MFFS` and `__MTFSF` functions read and set, respectively, the FPSCR register. The `__setflm` function is provided as a convenience to set the FPSCR register and return the previous value.

Lowercase versions of the intrinsic functions names are provided as synonyms.

```
unsigned int __DIR(void);  
unsigned int __EIR(void);
```

Like `__DI()` and `__EI()`, these functions disable and enable interrupts. In addition, they return an `unsigned int` key that represents the state of interrupts prior to their operation.

The compiler does not output any synchronization instructions before or after these intrinsics. Depending on the state of the processor and caches, you may need to insert synchronization instructions manually.

```
unsigned int __GETSR(void);
void __SETSR(unsigned int);
```

The `__GETSR` function returns the current value of the *Machine State Register* (MSR).

The `__SETSR` function takes a 32-bit integer argument and stores it into this register.

The compiler does not output any synchronization instructions before or after these intrinsics. Depending on the state of the processor and caches, you may need to insert synchronization instructions manually.

```
int __abs(int x);
long __labs(long x);
double __fabs(double x);
float __fabsf(float x);
double __fnabs(double x);
float __fnabsf(float x);
```

The `__abs`, `__labs`, `__fabs`, and `__fabsf` functions return the absolute value of an integer, long, double, and float, respectively. The `__fnabs` and `__fnabsf` return the negation of the absolute value of a double or float, respectively.

```
__lhbrx(short *ra, short rb);
__lwbrx(int *ra, int rb);
void __sthbrx(short *ra, short rb);
void __stwbrx(int *ra, int rb);
```

The `__lhbrx` and `__lwbrx` functions generate the corresponding byte-reversed load instruction. Note that these are macros which set their second argument to the result of the load. The `__sthbrx` and `__stwbrx` functions generate the corresponding byte-reversed store instruction.

```
int __RLWINM(int rs, int sh, int mb, int me);
int __rlwinm(int rs, int sh, int mb, int me);
int __RLWNM(int rs, int rb, int mb, int me);
int __rlwnm(int rs, int rb, int mb, int me);
int __RLWIMI(int ra, int rs, int sh, int mb, int me);
int __rlwimi(int ra, int rs, int sh, int mb, int me);
```

The `__RLWINM`, `__RLWNM` and `__RLWIMI` functions insert an integer rotate instruction of the same name. The *sh*, *mb*, and *me* parameters must be constants in the range of 0-31.

Lowercase versions of the intrinsic functions names are provided as synonyms.

```
void __DCBF(void * ra, int rb);
void __dcbf(void * ra, int rb);
void __DCBT(void * ra, int rb);
void __dcbt(void * ra, int rb);
void __DCBST(void * ra, int rb);
void __dcbst(void * ra, int rb);
void __DCBZ(void * ra, int rb);
void __dcbz(void * ra, int rb);
```

The `__DCBF`, `__DCBT`, `__DCBST`, and `__DCBZ` functions insert a data cache control instruction of the same name.

Lowercase versions of the intrinsic functions names are provided as synonyms.

```
double __FMADD(double fra, double frc, double frb);
double __fmadd(double fra, double frc, double frb);
double __FMSUB(double fra, double frc, double frb);
double __fmsub(double fra, double frc, double frb);
double __FNMADD(double fra, double frc, double frb);
double __fnmadd(double fra, double frc, double frb);
double __FNMSUB(double fra, double frc, double frb);
double __fnmsub(double fra, double frc, double frb);
float __FMADDS(float fra, float frc, float frb);
float __fmadds(float fra, float frc, float frb);
float __FMSUBS(float fra, float frc, float frb);
float __fmsubs(float fra, float frc, float frb);
float __FNMADDS(float fra, float frc, float frb);
float __fnmadds(float fra, float frc, float frb);
float __FNMSUBS(float fra, float frc, float frb);
float __fnmsubs(float fra, float frc, float frb);
```

The `__FMADD`, `__FMSUB`, `__FNMADD`, `__FNMSUB`, `__FMADDS`, `__FMSUBS`, `__FNMADDS`, and `__FNMSUBS` functions insert the corresponding floating point multiply-and-add or multiply-and-subtract instruction.

Lowercase versions of the intrinsic functions names are provided as synonyms.

```
float __FSEL(float fra, float frc, float frb);
float __FRES(float fra);
float __FRSQRTTE(float fra);
```

These intrinsics correspond directly to the underlying hardware instructions.

Arithmetic Operation Instructions

```
extern unsigned int __MULUH(unsigned int a, unsigned int b);
extern unsigned int __mulhwu(unsigned int a, unsigned int b);
extern signed int __MULSH(signed int a, signed int b);
extern signed int __mulhw(signed int a, signed int b);
```

The `__MULUH(a,b)` function takes as arguments two 32-bit unsigned integers *a* and *b* and returns the high 32-bit half of their 64-bit unsigned product.

The `__MULSH(a,b)` function takes as arguments two 32-bit signed integers *a* and *b* and returns the high 32-bit half of their 64-bit signed product.

The `__mulhwu` and `__mulhw` functions are provided as synonyms for `__MULUH` and `__MULSH`, respectively.

```
unsigned int __CLZ32(unsigned int a); /* zeros */
unsigned int __cntlzw(unsigned int a); /* zeros */
```

Takes a 32-bit integer argument and returns the count of leading zeros, which is a number from 0 to 32.

The `__cntlzw` function is provided as a synonym for `__CLZ32`.

Paired Single Extensions

The Paired Single Extensions enable support for a new data type, `f32x2` (or `__vec2x32float__`), which represents a vector of two 32-bit floating point numbers. When possible, variables with this type are allocated to the floating point registers. The floating point values can be operated on in parallel using the intrinsics described below.

The `f32x2` type supports brace delimited constant initializers similarly to arrays or structures in C. For example, the following code initializes the PS0 and PS1 fields of `t` to 1.0 and -1.0.

```
f32x2 t = { 1.0f, -1.0f };
```

As in C, the following example initializes the PS0 and PS1 fields of `t` to 1.0 and 0.0.

```
f32x2 t = { 1.0f };
```

The PS0 and PS1 fields of the vector can be read and written as if the vector was an array of fixed size using square brackets. The following example returns the sum of the PS0 and PS1 halves of the parameter *a*.

```
float func(f32x2 a) { return a[0] + a[1]; }
```

The Paired Single type is passed in registers following the same conventions as floating point arguments. Similarly, the PS1 halves are saved and restored from the stack along with the PS0 halves. See “Register Usage” on page 105 for details.

Operations not listed above are not permitted with the `f32x2` data type. For example, a Paired Single variable may not be cast to another type, added to another Paired Single with the `+` operator, or negated using the unary `-` operator.

Prototypes for the Paired Single Extensions can be found in `install_dir/include/ppc/ppc_ps.h` and can be included as follows:

```
#include <ppc_ps.h>
```

```
f32x2 __PS_ABS(f32x2);
f32x2 __PS_NABS(f32x2);
f32x2 __PS_NEG(f32x2);
f32x2 __PS_RES(f32x2);
f32x2 __PS_RSQRTE(f32x2);
```

These intrinsics compile directly to the underlying instruction of the same name.

```
f32x2 __PS_ADD(f32x2, f32x2);
f32x2 __PS_DIV(f32x2, f32x2);
f32x2 __PS_MERGE00(f32x2, f32x2);
f32x2 __PS_MERGE01(f32x2, f32x2);
f32x2 __PS_MERGE10(f32x2, f32x2);
f32x2 __PS_MERGE11(f32x2, f32x2);
f32x2 __PS_MUL(f32x2, f32x2);
f32x2 __PS_MULS0(f32x2, f32x2);
f32x2 __PS_MULS1(f32x2, f32x2);
f32x2 __PS_SUB(f32x2, f32x2);
```

These intrinsics compile directly to the underlying instruction of the same name.

```
f32x2 __PS_MADD(f32x2, f32x2, f32x2);
f32x2 __PS_MADDS0(f32x2, f32x2, f32x2);
f32x2 __PS_MADDS1(f32x2, f32x2, f32x2);
f32x2 __PS_MSUB(f32x2, f32x2, f32x2);
f32x2 __PS_NMADD(f32x2, f32x2, f32x2);
f32x2 __PS_NMSUB(f32x2, f32x2, f32x2);
f32x2 __PS_SEL(f32x2, f32x2, f32x2);
f32x2 __PS_SUM0(f32x2, f32x2, f32x2);
f32x2 __PS_SUM1(f32x2, f32x2, f32x2);
```

These intrinsics compile directly to the underlying instruction of the same name.


```
f32x2 __PSQ_L(const void *, int W, int I);
f32x2 __PSQ_LX(const void *, int offset, int W, int I);
void __PSQ_ST(void *, f32x2, int W, int I);
void __PSQ_STX(void *, int offset, f32x2, int W, int I);
```

Loading and storing of vectors can be accomplished using standard C pointer or array syntax. To make use of the *W* or *I* parameters, the above intrinsics must be used. The *W* and *I* parameters must be compile time constants of 0 or 1, and 0 to 7, respectively. Quantization registers *GQR0* and *GQR1* are reserved for the compiler and should not be written. Registers *GQR2* through *GQR4* are used by the operating system. Refer to the operating system documentation before modifying. The remaining quantization registers can be used by the user program.

To read and write *GQR* registers, the general read or write *SPR* intrinsics can be used. The user mode accessible *GQR* registers are numbered 896 through 903.

```
unsigned int __MFSPR(unsigned int spr);
void __MTSPR(unsigned int spr, unsigned int val);
```

```
f32x2 __PS_FDUP(float);
```

Returns a vector with the parameter duplicated in both halves. With optimizations enabled, this intrinsic can execute in zero cycles depending upon how its parameter was created.

```
f32x2 __PS_MADDS0F(f32x2, float, f32x2);
f32x2 __PS_MULS0F(f32x2, float);
f32x2 __PS_SUM0F(float, f32x2, f32x2);
f32x2 __PS_SUM1F(float, f32x2, f32x2);
f32x2 __PS_SUM1_F(f32x2, float, f32x2);
f32x2 __PS_SUM1FF(float, float, f32x2);
```

These intrinsics provide access to paired single instructions in cases where the *PS1* half of one (or more) of the arguments is not read. In these cases, a single float can be passed instead. For *__PS_SUM1*, note that the name of the intrinsic changes depending upon which arguments are passed as floats.

```
f32x2 __PS_NEGS(float);
f32x2 __PS_RESS(float);
f32x2 __PS_ADDS(float, float);
f32x2 __PS_SUBS(float, float);
f32x2 __PS_MULS(float, float);
f32x2 __PS_DIVS(float, float);
f32x2 __PS_MADDS(float, float, float);
f32x2 __PS_NMADDS(float, float, float);
f32x2 __PS_MSUBS(float, float, float);
f32x2 __PS_NMSUBS(float, float, float);
```

These intrinsics are provided for backwards compatibility with previous versions of the tools. In general, *__PS_FDUP* should be used instead. These intrinsics perform a standard floating point instruction which duplicates its result into the two halves of a paired single.

Four Wide Vectors with Paired Singles

When porting applications from other platforms, it may be necessary to implement a vector of four floating point numbers. This section provides guidelines on how to best accomplish this task. Using this type is only recommended to ease porting. Better run-time performance can often be achieved using paired singles directly.

To store the four floats in a class, use either two individual paired single members or an array of two paired singles. Do not place these members in a union with other data types. Using unions will inhibit some type based optimizations, resulting in worse performance.

To operate on the vector, avoid accessing the individual floats whenever possible. Use the paired single intrinsics described above. In the cases where accessing the floats are unavoidable, use the paired single array subscript operator. Unless the paired singles are in an array, do not take the address of the first member and use pointer arithmetic to access the second member, as in the example below. This has undefined semantics, and may result in unexpected behavior in some cases.

```
class Vect4 {
    f32x2 a, b;
    /* Do not write an accessor like this: */
    float get_undefined(int i) { return ((float *) &a)[i]; }
};
```

Be aware when porting algorithms from other platforms that four wide vectors are not a native data type. It will not be allocated to a register, though sometimes optimizations will place halves in registers. For algorithms that use lots of out of line function calls, be aware that four wide vector variables passed by value will be passed in memory, not registers. Algorithms may have been written assuming that a large number of floats can be simultaneously stored in registers. The paired single register set is more limited, which may result in more temporaries being placed in memory.

In general, because four wide vectors do not fit in registers, it is recommended that they be passed and returned to functions by pointer or reference. This may not be possible in all cases due to an existing interface.

The following sample implementation of a four wide vector class follows the tips above.

```
#include <ppc_ps.h>
class Vec4 {
    f32x2 a, b;

public:
    Vec4(f32x2 x, f32x2 z) : a(x), b(z) { }

    Vec4(float x, float y, float z, float w) {
        a[0] = x;
        a[1] = y;
        b[0] = z;
        b[1] = w;
    }

    Vec4(float x) {
        a = b = __PS_FDUP(x);
    }

    float get(int i) const {
        switch (i) {
            case 0: return a[0];
            case 1: return a[1];
            case 2: return b[0];
            case 3: return b[1];
        }
        return a[0];
        /* NOT: return ((float *) &a)[i] */
    }

    void abs(Vec4 &ret) const {
        ret.a = __PS_ABS(a);
        ret.b = __PS_ABS(b);
    }

    Vec4 abs_bad(void) const {
        /* Return by value - not recommended for performance. */
        return Vec4(__PS_ABS(a), __PS_ABS(b));
    }

    void add(Vec4 &ret, Vec4 const &x) const {
        ret.a = __PS_ADD(a, x.a);
        ret.b = __PS_ADD(b, x.b);
    }

    Vec4 add_bad(Vec4 x) const {
        /* Return by value - not recommended for performance. */
        return Vec4(__PS_ADD(a, x.a), __PS_ADD(b, x.b));
    }
};
```


Libraries and Header Files

Chapter 17

This Chapter Contains:

- The Green Hills Header Files and Standard Libraries
- Advanced Library Topics
- Green Hills Standard C Library Functions
- Customizing the Run-Time Environment Libraries and Object Modules

This chapter describes the standard language libraries provided with the Green Hills tools for PowerPC. Libraries are files which contain collections of object files.

Green Hills provides libraries to implement language features, such as the Standard C libraries, the C++ libraries, and various aspects of run-time support. These libraries, and any libraries that you create, are linked into the program image by the linker, **elxr**.

Header files provide declarations and macros that may be used while programming. Many of the functions in the standard libraries are declared in header files. In C and C++, header files are referenced in the source code by using a `#include` directive, which is read by the compiler's preprocessor.

By convention, libraries have a **.a** extension, and are named in the form **libname.a**. C language header files generally have a **.h** extension, and C++ headers either have a **.h** extension or no extension.

This chapter documents Green Hills libraries. For information about creating and maintaining your own libraries, see Chapter 12, "The ax Librarian".

For more information, see:

- P. J. Plauger, *The Standard C Library* (describes the C89 library)
- Harbison & Steele, *C: A Reference Manual (5th Ed.)* (describes the C99 library)
- B. Stroustrup, *The C++ Programming Language (3rd Ed.)*
- P. J. Plauger, *Dinkum C++ Library Reference* (included with your distribution, at `install_dir/scxx/html/index.html`)
- P. J. Plauger, *Dinkum Abridged Library Reference* (for EC++ and EEC++; included with your distribution, at `install_dir/ecxx/html/index.html`).



Note Your Green Hills compiler license permits you to make unlimited distributions of programs linked with the Green Hills library object code without charge. Distribution of Green Hills library source code or object code is not permitted.

The Green Hills Header Files and Standard Libraries

Your Green Hills tools for PowerPC provide a number of directories that contain header files and standard libraries for the C and C++ languages, together with a number of other low-level system libraries.

C and C++ Header File Directories

Depending upon the language you are compiling, the driver selects one or more of the following directories for header files which are specified in `#include` preprocessor directives:

- **include/ppc** — Contains header files for PowerPC extensions for C and C++.
- **ansi** — Contains Standard C library header files.
- **scxx** — Contains Standard C++ library header files.
- **eecxx** — Contains Extended Embedded C++ library header files.
- **ecxx** — Contains Embedded C++ library header files.

Library Directories

Depending upon the target architecture that you specify and whether or not you are using INTEGRITY, the driver selects one of the following directories (inside *install_dir/lib*) for libraries to resolve references in the header files. Each directory contains a library set optimized for use with a specific type of target CPU in the PowerPC processor family.

- **ppc** (Default) — Selected for big endian targets with Hardware Floating Point.

To determine which directory your particular target defaults to, see “PowerPC Processor Variants” on page 114.

Libraries

Each library directory contains a version of the following set of libraries, built for the particular target architecture:

- C Libraries (these libraries are always searched unless you pass the **-nostdlib** driver option):
 - **libansi.a** — Standard C library. This is the primary C library.
 - **libmath.a** — Standard Math library. This library is searched after **libansi.a**.
 - **libnoflt.a** — Provides versions of `printf`, `scanf`, and other functions, which exclude floating point support, and so are smaller than those in **libansi.a**. This library is searched before **libansi.a**, and is only used if the **-fnone** or **-nofloatio** driver options are specified.

- C++ Libraries (see also “Specifying C++ Libraries” on page 623):

C++ libraries are searched if there is a C++ source file on the driver command line or if you use the C++ driver, **cxppc**. If C++ libraries are present, they are searched before C libraries.

- **libscnoe.a** — Standard C++ Library without exceptions.
 - **libsce.a** — Standard C++ Library with exceptions.
 - **libeecnoe.a** — Extended embedded C++ Library without exceptions.
 - **libeece.a** — Extended embedded C++ Library with exceptions.
 - **libecnoe.a** — Embedded C++ Library without exceptions. This library is a subset of **libeecnoe.a**.
 - **libece.a** — Embedded C++ Library with exceptions.
 - **libsedgnoe.a** — Run-time support for standard C++, without exceptions.
 - **libsedge.a** — Run-time support for standard C++, with exceptions.
 - **libedgnoe.a** — Run-time support for embedded and extended embedded C++, without exceptions.
 - **libedge.a** — Run-time support for embedded and extended embedded C++, with exceptions.
- Low-level System Libraries (These libraries are always searched unless you pass the **-nostdlib** driver option):
 - **libind.a** — Language-independent library, containing support routines for features such as software floating point, run-time error checking, C99 complex numbers, and some general purpose routines of the ANSI C library.

- **libstartup.a** — Run-time environment startup routines. The source code for the modules in this library is provided in the **src/libstartup** directory (see “Low-Level Startup Library libstartup.a” on page 755).
- **libsys.a** — Run-time environment system routines. The source code for the modules in this library is provided in the **src/libsys** directory (see “Low-Level System Library libsys.a” on page 756).
- **libarch.a** — Target-specific run-time support.
- Other Libraries:
 - **libdbmem.a** — Provides a version of `malloc` with memory-checking. This library is selected when you pass either the **-check=alloc** or **-check=memory** driver options.
 - **libmulti.a** — Supports procedure calls from the MULTI Debugger command line. This library is selected when you pass the **-G** driver option.

Automatic Searching of Libraries

The compiler chooses libraries to search based on the options which you pass. By default, for a project written in C, it will:

- Search for header files in the directories **include/ppc** and **ansi**.
- Search for libraries in the **lib/ppc** directory, and search for symbols in the following libraries, in this order:
 - **libansi.a** (the standard C library)
 - **libmath.a** (to obtain necessary math routines)
 - **libind.a** (to obtain miscellaneous C routines and software floating point routines)
 - **libstartup.a** (to obtain run-time startup routines)
 - **libsys.a** and **libarch.a** (to obtain run-time support routines)

When you specify a particular target to build for, or pass other options, the compiler may change its search paths, and select additional libraries to link against. For example:

```
cxppc --eel -G -fnone -bsp=sand7400 test.cxx
```

This command instructs the compiler to:

- Search for header files in the directories **eecxx**, **include/ppc**, and **ansi**, because **--eel** specifies extended embedded C++ (see “Specifying C++ Libraries” on page 623).
- Search for libraries in the **ppc7400** directory (because of the option **-bsp=sand7400**, which specifies the Motorola Sandpoint 7400 board, containing a PowerPC 7400 processor).
- Open the following libraries for linking against in this order:
 - **libmulti.a** (because of the **-G** option)
 - **libeecnoe.a** (because of the **--eel** option)
 - **libedgnoe.a** (because of the **--eel** option)
 - **libnoflt.a** (because of the **-fnone** option)
 - **libansi.a**, **libmath.a**, and **libind.a** (default C libraries)
 - **libstartup.a**, **libsys.a**, and **libarch.a** (default lower level libraries)



Note For information about specifying your own libraries and headers to search for, see “Using Your Own Header Files and Libraries” on page 122.

Advanced Library Topics

Less Buffered I/O

The Green Hills ANSI C library includes a *Standard I/O Package* (“**stdio**”), which provides formatted I/O using `printf()` and `scanf()`, character I/O using `getc()` and `putc()`, and other features.

If I/O is completely *unbuffered*, every character read or written requires a system call. This unbuffered I/O is slow, but it ensures that I/O is not delayed until the buffer is full or lost by being left in a buffer if the application exits abnormally.

As a consequence, in traditional implementations, most I/O performed in **stdio** is *fully buffered*. This improves performance by increasing the average number of characters written per system call, but requires additional space for the code to manage the buffers, as well as for the buffers themselves. Often, several kilobytes of code are added to the application, because the **stdio** package invokes `malloc()` and other routines to manage the buffering.

In order to provide a reasonable compromise between fully buffered and unbuffered I/O, the Green Hills C library defaults to a *less buffered I/O* mode. Rather than allocate a buffer for each file via `malloc()` (which is used as long as the file is open), buffering is performed within `fprintf()`, `fwrite()`, `fputs()`, `puts()`, `printf()`, `vprints()`, `vfprintf()`, and their wide character forms. Because of this approach, none of the I/O functions call `malloc()`, possibly reducing the size of the overall program by a few kilobytes.

During a single call, any characters written to one of these functions are buffered in a buffer on the stack. One function call often requires only one output system call. After the function completes, all characters are written to the file. No characters are left in a buffer after the function call, eliminating both the risk of output loss and the need to flush buffers when a file is closed or the program exits. No input routines other than `fread()` are buffered by the less buffered I/O method. Files are not closed upon program termination, except as noted below.

You can enable full buffering of any file by calling either `setbuf()` or `setvbuf()`. In this case, characters are only written to a buffered file when the buffer is full or when `fflush()` or `fclose()` is called. Upon normal termination of the program, if `setbuf()` or `setvbuf()` has been called at any time, all open files will be flushed and then closed.

You can enable line buffering by calling `setvbuf()` with `_IOLBF` as the third parameter. Line buffering is the same as full buffering with the exception that `fflush()` is called at the end of `fputs()` and all forms of `printf()`.

All files are processed in exactly the same manner, whether they are opened by default (**stdin**, **stdout**, and **stderr**) or by `fopen()`.

Buffering of C++ I/O Streams

The C++ I/O streams are implemented on top of the C **stdio** files. When you use C++ I/O streams, the constructors for these streams enable full buffering for files, and line buffering for **stdout** and **stderr**.

To disable this buffering, call `pubsetbuf()`, `setbuf()`, or `setvbuf()` before outputting anything onto either of the I/O streams. For example:

```
cout.rdbuf()->pubsetbuf(0, 0);
```

or

```
setbuf(stdout, NULL);
```

If you call one of these functions after outputting anything onto an I/O stream, buffering will remain unchanged, in accordance with the C specification for `setvbuf()`.

64-bit Integer Arguments and printf

The type `long long` was an extension to ANSI C which has become official in the ISO C99 standard. Since this type is larger than either `int` or `long`, it requires special handling in the `printf` and `scanf` routines.

As in earlier versions of `printf` and `scanf`, a single 'l' modifier indicates an argument of type `long`. For example:

```
printf("large number: %ld", 0x7fffffffL);
```

However, when the 'l' modifier is repeated, an argument of type `long long` is indicated. For example:

```
printf("very large number: %lld", 0x7fffffffffffffffffLL);
```

For backwards compatibility with older implementations, `printf` and `scanf` also allow the 'L' modifier to indicate an argument of type `long long`, but this is non-standard and is not recommended.

Memory Allocation and `alloca()`

The `alloca()` function provides a mechanism to allocate a chunk of memory that is automatically freed when the current function exits. The amount of memory allocated does not need to be a compile-time constant. Traditionally, `alloca()` is implemented by subtracting the stack pointer by enough memory to accommodate the requested amount of memory, plus any necessary alignment or padding. The function return sequence automatically accounts for this, so that the memory is freed as a result of leaving the function. Other implementations exist, but they typically involve compromises that may either affect the performance of the system or how soon the requested memory is actually reclaimed.

You must include **`alloca.h`** to access `alloca()` and its two related functions:

```
void *alloca(size_t size);
```

Allocates *size* bytes of memory with the same alignment restrictions observed by `malloc()`. The memory is automatically freed when the caller exits.

```
void *__get_stack_pointer(void);
```

Returns the current value of the stack pointer. This value is only intended to be used with `__set_stack_pointer()`.

```
void __set_stack_pointer(void *sp);
```

Restores the stack pointer to *sp*, which must have been returned by a prior call to `__get_stack_pointer()`. Memory allocated by `alloca()` between the `__get_stack_pointer()` and `__set_stack_pointer()` calls is popped from the stack and may no longer be used. Similarly, other intermediate values returned by `__get_stack_pointer()` that now point into freed stack space are stale and should not be used later with `__set_stack_pointer()`.

The pair of `__get_stack_pointer()` and `__set_stack_pointer()` calls must occur within the same function (and in the same function invocation), and they must be in the same block scope, or the `__set_stack_pointer()` call may be inside a scope nested within the scope of `__get_stack_pointer()`.

Any other use of `__set_stack_pointer()` has undefined behavior and may corrupt the system.



Note On some targets, C99 variable-length arrays (VLAs) are implemented using the `alloca()` mechanism. Declaring a VLA in a function may interfere with use of the function calls in this table.

If you use `alloca()` in a function and do not need to free the memory it allocates until the function returns, you do not need to save or recall the stack pointer. However, if you need to free the memory before the function returns (for example, when looping over a block that dynamically allocates memory), save the stack pointer before calling `alloca()` and recall it when you no longer need the allocated memory. For example:

```
#include <stdio.h>
#include <alloca.h>

void printAlphabet(int length)
{
    char *buffer = alloca(length + 1);
    int i;
    for (i=0; i < length; ++i) {
        buffer[i] = 'a' + i;
    }
    buffer[i] = '\0';
    printf("%s\n", buffer);
    /* Because printAlphabet() restores the stack pointer as part of
     * the function epilogue, you do not need to free "buffer" manually.*/
}

int main()
{
    int length;
    void *old_sp;
    for (length=1; length < 27; ++length) {
        /* Print the alphabet up to "length" */
        printAlphabet(length);
    }
    /* Print the same sequence without calling printAlphabet().
     * When using alloca() in a loop, get the stack pointer so that
     * you can restore it at the end of the loop */
    old_sp = __get_stack_pointer();
    for (length=1; length < 27; ++length) {
        char *alphabet = alloca(length + 1);
        int i;
        for (i=0; i < length; ++i) {
            alphabet[i] = 'a' + i;
        }
        alphabet[i] = '\0';
        printf("%s\n", alphabet);

        /* Return the stack pointer to the location it was in before
         * calling alloca(). This releases the allocated memory */
        __set_stack_pointer(old_sp);
    }
}
```

Memory Allocation and malloc()

`malloc()`, `calloc()`, `free()`, and the C++ `new` and `delete` operators are all examples of dynamic memory allocation. Source code for the Green Hills implementation of dynamic memory allocation is not provided, although most memory allocation problems can be easily debugged. First, ensure that your implementation of `ind_heap.c` is appropriate for your system. Most other memory allocation issues come down to a case where the user code does something illegal with dynamic memory, such as:

- Writing before the beginning or after the end of a piece of dynamic memory.
- Accessing memory that has been deallocated by calling `free()` or the C++ `delete` operator.
- Deallocating memory more than once.
- Deallocating memory that was not dynamically allocated (that is, global, static, or local memory).

These problems may often be tracked down by using General or Intensive Run-Time Memory checks (see “Run-Time Memory Checks” in “Debugging Options” on page 187, or “Chapter 13, Viewing Memory Allocation Information” in the *MULTI: Debugging* book).

Using Thread-Safe Library Functions

A *thread-safe* or *reentrant* function executes correctly in a multithreaded environment, even if the function is called by multiple threads concurrently or if it is interrupted or suspended and then resumed. A *thread* is a single task running in a multitasking environment in which several tasks share the same memory space.

Some Green Hills library functions are not inherently thread-safe. The libraries provide four mechanisms for an application to achieve thread-safe behavior. The following sections describe the four mechanisms and, where necessary, explain how to port them to a multithreaded environment.

- “Using a Lock to Ensure Critical Code Is Thread-Safe” on page 716
- “Using File Locks to Ensure I/O Is Thread-Safe” on page 716
- “Allocating Space in Each Thread for Static Data” on page 717
- “Using Alternate Thread-Safe Library Functions” on page 719

In some cases, the library provides more than one way to achieve thread-safe behavior. For example, you can make calls to `strtok()` thread-safe either by allocating space in each thread for the static data it uses, or by changing the code to use the inherently thread-safe `strtok_r()` function. The appropriate choice is left up to you.

For applications that require maximum I/O performance, Green Hills provides versions of some I/O library functions that are not thread-safe. These functions have names ending in `_unlocked`, such as `putc_unlocked()` and `getc_unlocked()`. Applications using these functions must guarantee that no two threads concurrently access the same file.

Porting to a new multithreaded environment involves customizing the following functions, located in `src/libsys/ind_lock.c`, `src/libsys/ind_errn.c`, and `src/libsys/ind_thr.c`.

```
void __ghsLock(void);
```

Acquires global lock. Blocks until the lock becomes available.

```
void __ghsUnlock(void);
```

Releases global lock.

```
void __gh_lock_init(void);
```

Initializes the global lock data structure before it is used.

```
void __ghs_flock_file(void *addr);
```

Acquires per-file lock **addr**. Blocks until the lock becomes available.

```
void __ghs_funlock_file(void *addr);
```

Releases per-file lock **addr**.

```
int __ghs_ftrylock_file(void *addr);
```

Attempts to acquire per-file lock **addr**. Returns 0 on success and nonzero if the lock could not be acquired. May return -1 if this behavior cannot be implemented. This function must not block.

```
void __ghs_flock_create(void **addr);
```

```
void __ghs_flock_destroy(void *addr);
```

Initializes and cleans up per-file lock data structure. The `__ghs_flock_create()` implementation may store a pointer in `addr`. That pointer is passed to the other per-file lock calls.

```
void *__ghs_GetThreadLocalStorageItem(int specifier);
```

Retrieves pointer to specified per-thread data item. May return NULL if the data item is not allocated for each thread.

For information about the default implementation of per-thread storage and locks in Green Hills supported operating systems, see the appropriate RTOS User's Guide (*INTEGRITY Libraries and Utilities User's Guide* or *u-velOSity User's Guide*).

Using a Lock to Ensure Critical Code Is Thread-Safe

Code that accesses resources such as global data structures, hardware devices, or memory shared between two or more threads is *critical code*. These shared resources may become corrupted when two or more threads write to the same data structures simultaneously, or when one thread reads data while another thread is writing to that same data.

To make critical code thread-safe, you must implement a global, recursive lock mechanism that allows a function to completely execute critical code before that code is accessed by another thread. Green Hills library functions call `__ghsLock()` before critical code and `__ghsUnlock()` after critical code. The `__ghsLock()` function must acquire the global lock or block the calling thread if any other thread has already acquired the global lock. The `__ghsUnlock()` function releases the global lock. These functions operate recursively. If `__ghsLock()` is called n times, the lock is released only after `__ghsUnlock()` is called n times.

Using File Locks to Ensure I/O Is Thread-Safe

You can also implement recursive locking for file I/O to ensure thread safety. If multiple threads are performing I/O on the same file concurrently, the file may become corrupted. This section discusses three possible implementations of file locking, from the simplest to the most flexible.

The simplest file lock implementation uses the global lock by having `__ghs_flock_file()` call `__ghsLock()` and `__ghs_funlock_file()` call `__ghsUnlock()`. However, this implementation may cause slow performance if one thread holds the global lock until it completes relatively slow I/O while other threads are attempting to use library functions that contain critical sections.

Another file lock implementation choice is to use a separate global lock for I/O. This avoids locking the entire library during I/O. However, performance

concerns could still arise if other threads are attempting to perform I/O and must wait for the single global I/O lock.

The most flexible file lock implementation choice is to create a separate lock for each file. To implement a lock for each file, have `__ghs_flock_create()` initialize a lock, and use the argument to store a pointer to the lock data structure. The `__ghs_flock_file()` and `__ghs_funlock_file()` functions have the pointer passed to them, so they can locate the specific lock for each file. Finally, the `__ghs_flock_destroy()` function should clean up and deallocate the lock.

Allocating Space in Each Thread for Static Data

This section describes how you can make certain library functions thread-safe by allocating separate copies of the data structures they use in each thread. The data structures are then referred to as *per-thread data* or *thread local storage*.

The additional space needed to make all such library functions completely thread-safe is several hundred bytes per thread. Therefore, the decision of which, if any, of these data structures are allocated for each thread is left to the implementation. In many cases, a more practical solution would be to use alternate library functions that are thread-safe. For more information, see “Using Alternate Thread-Safe Library Functions” on page 719.

An implementation could allow some library functions to operate on global data. It could either restrict all such function calls to a single thread or allow the application to synchronize access by using a mutual exclusion mechanism around the function call and the use of the return value. For example, if two threads wanted to call `asctime()`, the function call and the use of the return value could be protected by first calling `__ghsLock()` and afterwards calling `__ghsUnlock()`.

The function `__ghs_GetThreadLocalStorageItem()` in **src/libsys/ind_thr.c** returns a pointer to the specified per-thread data item, or NULL. Returning NULL causes the library to use a single, global data item. The default implementation for stand-alone programs returns NULL for all but one such request.

If you implement a per-thread `errno` in `__ghs_GetThreadLocalStorageItem()`, you must

define the preprocessor symbol `USE_THREAD_LOCAL_ERRNO` in `src/libsys/ind_errn.c`.

The data item pointers `__ghs_GetThreadLocalStorageItem()` can return are as follows:

Specifier (Value)	Example Data Item Declaration
<code>__ghs_TLS_asctime_buff(0)</code>	<code>char asctime_buff[30]</code>
<code>__ghs_TLS_tmpnam_space(1)</code>	<code>char tmpnam_space[L_tmpnam]</code>
<code>__ghs_TLS_strtok_saved_pos(2)</code>	<code>char *strtok_saved_pos</code>
<code>__ghs_TLS_Errno(3)</code>	<code>int errno</code>
<code>__ghs_TLS_gmtime_temp(4)</code>	<code>struct tm gmtime_temp</code>
<code>__ghs_TLS___eh_globals(5)</code>	<code>void *__eh_globals</code>
<code>__ghs_TLS_SignalHandlers(6)</code>	<code>SignalHandler</code> <code>SignalHandlers[_SIGMAX]</code>

Upgrading Earlier `GetThreadLocalStorage()` Implementations

In versions of MULTI prior to 4.2.3, the function `GetThreadLocalStorage()`, declared in `ind_thrd.h`, was used to retrieve thread-specific or global library data structures. If you previously customized `GetThreadLocalStorage()` for your environment, the simplest way to upgrade is to add the following implementation of `__ghs_GetThreadLocalStorageItem()`.

```
enum __ghs_ThreadLocalStorage_specifier
{
    __ghs_TLS_asctime_buff,
    __ghs_TLS_tmpnam_space,
    __ghs_TLS_strtok_saved_pos,
    __ghs_TLS_Errno,
    __ghs_TLS_gmtime_temp,
    __ghs_TLS___eh_globals,
    __ghs_TLS_SignalHandlers
};

void *__ghs_GetThreadLocalStorageItem(int specifier)
{
    ThreadLocalStorage *tls = GetThreadLocalStorage();
    if(!tls)
        return (void *)0;
    switch (specifier)
    {
        case (int)__ghs_TLS_Errno:
            return (void *)&tls->Errno;
        case (int)__ghs_TLS_SignalHandlers:
            return (void *)&tls->SignalHandlers;
        case (int)__ghs_TLS_asctime_buff:
            return (void *)&tls->asctime_buff;
        case (int)__ghs_TLS_tmpnam_space:
            return (void *)&tls->tmpnam_space;
        case (int)__ghs_TLS_strtok_saved_pos:
            return (void *)&tls->strtok_saved_pos;
        case (int)__ghs_TLS_gmtime_temp:
            return (void *)&tls->gmtime_temp;
        case (int)__ghs_TLS___eh_globals:
            return (void *)&tls->__eh_globals;
    }
    return (void *)0;
}
```

Using Alternate Thread-Safe Library Functions

Green Hills provides alternatives for some library functions that are not inherently thread-safe. These alternate, thread-safe functions have names ending in `_r`, such as `asctime_r()` and `rand_r()`. For example, you can use `rand_r()` to replace calls to `srand()` and `rand()`.

Green Hills does not provide thread-safe versions of the obsolete library functions `ecvt()`, `fcvt()`, and `gcvt()`, and may remove these functions

in a future release. Green Hills recommends replacing calls to these functions with appropriate calls to `sprintf()`.

Customizing Thread-Safe Library Functions

Green Hills provides source code for some low-level functions in the **src/libsys** and **src/libstartup** directories, so you can customize them for your particular environment. When modified, these functions must remain thread-safe so that functions in the Green Hills libraries that call them also remain thread-safe. For a list of these functions, see “Green Hills Standard C Library Functions” on page 721.

Green Hills Standard C Library Functions

The functions documented in the ISO C99 Standard are contained in the **libansi.a**, **libmath.a**, **libind.a**, and **libstartup.a** libraries. You can find lists of these functions, as well as thread-safety information, in the following sections:

- “Standard C Functions in libansi.a” on page 722
- “libmath.a Functions” on page 738
- “libind.a Functions” on page 750
- “libstartup.a Functions” on page 750

The following code letters identify the level of thread-safety of the library functions listed in the remainder of this chapter.

Code	Thread Safety
Y	Completely thread-safe.
P	Partially thread-safe. You must implement a lock for complete thread safety.
I	Performs I/O or modifies I/O data structures. This is a special case of a partially thread-safe function.
T	Achieves thread safety by per-thread storage, so long as per-thread storage is implemented. See the preceding discussion on “Allocating Space in Each Thread for Static Data” on page 717
E	Writes to the global variable <code>errno</code> . Achieves thread safety by per-thread storage of <code>errno</code> .
N	Not thread-safe.
K	Safe to call from all INTEGRITY contexts.



Note Under INTEGRITY, all functions marked with a Y, P, I, T, or E are completely thread-safe. These functions are safe to call from within a Task, however, they may not be safe to call from an interrupt context unless they are also marked with a K.

Under u-velOSity, the thread-safety of functions marked with a P, I, T, or E depends on the kernel libraries you are using. Please refer to the *u-velOSity User's Guide* for more information.

Standard C Functions in **libansi.a**

This section lists functions available in **libansi.a** and the low-level functions in **libsys.a** on which these standard functions depend. The information is laid out in two tables:

- “libansi.a Standard C Functions and Dependencies” on page 722
- “libsys.a Function Implementation” on page 737

Depending on the operating system you are running on your target, and whether or not you are connecting to your target through a MULTI debug server, you may need to implement environment-specific behavior for the low-level functions in **libsys.a**. The **libsys.a** function implementation table describes how the Green Hills tools implement these functions by default.

libansi.a Standard C Functions and Dependencies

This table lists functions in **libansi.a**. The low-level functions in **libsys.a** on which each standard function depends are listed in the **Dependency** column. For information about whether or not you need to implement environment-specific behavior for a **libsys.a** function, see “libsys.a Function Implementation” on page 737 and “Low-Level System Library libsys.a” on page 756.

Function		Dependency	Thread Safety
void	<code>abort(void);</code>	<code>raise()</code>	Y
int	<code>abs(int x);</code>		Y
char	<code>*asctime(const struct tm *timeptr);</code>	<code>__gh_timezone()</code>	T
char	<code>asctime_r(const struct tm *timeptr, char *buffer);</code>	<code>__gh_timezone()</code>	Y
void	<code>_assert(const char *problem, const char *filename, int line);</code>	<code>raise()</code> <code>write()</code>	IE
void	<code>assert(int value);</code>	<code>raise()</code> <code>write()</code>	IE
int	<code>atexit(void (*func)(void));</code>		P
int	<code>atoi(const char *nptr);</code>		E
long	<code>atol(const char *nptr);</code>		E
long long	<code>atoll(const char *nptr);</code>		E

Function		Dependency	Thread Safety
int	<code>bcmp(char *b1, char *b2, int length);</code>		Y
void	<code>bcopy(char *from, char *to, int n);</code>		Y
void	<code>*bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));</code>		Y
wint_t	<code>btowc(int c);</code>		Y
void	<code>bufcpy(char *to, char *from, int n);</code> Obsolete. Will be removed in the next release.		Y
void	<code>bzero(char *pt, int n);</code>		Y
void	<code>*calloc(size_t nmemb, size_t size);</code>	<code>sbrk()</code>	P
void	<code>cfree(char *item);</code>	<code>sbrk()</code>	P
void	<code>clearerr(FILE *stream);</code>		I
void	<code>clearn(int n, char *pt);</code> Obsolete. Will be removed in the next release.		Y
clock_t	<code>clock(void);</code>	<code>times()</code>	Y
char	<code>*ctime(const time_t *timer);</code>	<code>__gh_timezone()</code> <code>localtime()</code>	T
char	<code>*ctime_r(const time_t *timer, char *buffer);</code>	<code>__gh_timezone()</code> <code>localtime()</code>	Y
double	<code>difftime(time_t time1, time_t time0);</code>		Y
div_t	<code>div(int numer, int denom);</code>		Y
char	<code>*ecvt(double value, int ndig, int *decpt, int *sign);</code> Obsolete. Will be removed in the next release.		N
int	<code>eprintf(const char *format, ...);</code>	<code>write()</code>	IE

Function		Dependency	Thread Safety
void	<code>exit(int status);</code>	<code>_Exit()</code>	P
int	<code>execl(const char *name, const char *arg0, ...);</code> Obsolete. Will be removed in the next release.		Y
int	<code>execle(const char *name, const char *arg0, ...);</code> Obsolete. Will be removed in the next release.		Y
int	<code>execv(const char *name, char *const *argv);</code> Obsolete. Will be removed in the next release.		Y
int	<code>fclose(FILE *stream);</code>	<code>close()</code> <code>write()</code>	IE
char	<code>*fcvt(double value, int ndig, int *decpt, int *sign);</code> Obsolete. Will be removed in the next release.		N
FILE	<code>*fdopen(int fno, const char *mode);</code>	<code>write()</code> <code>close()</code>	IE
int	<code>feof(FILE *stream);</code>		Y
int	<code>ferror(FILE *stream);</code>		Y
int	<code>fflush(FILE *stream);</code>	<code>write()</code>	IE
int	<code>ffs(int i);</code>		Y
int	<code>fgetc(FILE *stream);</code>	<code>read()</code>	IE
int	<code>fgetpos(FILE *restrict stream, fpos_t *restrict pos);</code>	<code>lseek()</code>	IE
char	<code>*fgets(char *restrict str, int n, FILE *restrict stream);</code>	<code>read()</code>	IE
wint_t	<code>fgetwc(FILE *stream);</code>	<code>read()</code>	IE
wint_t	<code>fgetwc_unlocked(FILE *stream);</code>	<code>read()</code>	NE
wchar_t	<code>*fgetws(wchar_t *restrict s, int n, FILE *restrict stream);</code>	<code>read()</code>	IE
void	<code>_filbuf(FILE *file);</code>	<code>read()</code>	IE

Function		Dependency	Thread Safety
int	<code>fileno(FILE *stream)</code>		Y
void	<code>filln(int n, char *pt, int fill);</code> Obsolete. Will be removed in the next release.		Y
void	<code>_flsbuf(int ch, FILE *file);</code>	<code>write()</code>	IE
FILE	<code>*fopen(const char *restrict filename, const char *restrict mode);</code>	<code>close()</code> <code>open()</code> <code>write()</code>	IE
int	<code>fprintf(FILE *restrict stream, const char *restrict format, ...);</code>	<code>write()</code>	IE
int	<code>fputc(int c, FILE *stream);</code>	<code>write()</code>	IE
int	<code>fputs(const char *restrict s, FILE *restrict stream);</code>	<code>write()</code>	IE
wint_t	<code>fputwc(wchar_t c, FILE *stream);</code>	<code>write()</code>	IE
wint_t	<code>fputwc_unlocked(wchar_t c, FILE *stream);</code>	<code>write()</code>	NE
int	<code>fputws(const wchar_t restrict *s, FILE *restrict stream);</code>	<code>write()</code>	IE
size_t	<code>fread(void restrict *ptr, size_t size, size_t nmemb, FILE *restrict stream);</code>	<code>read()</code>	IE
void	<code>free(void *ptr);</code>	<code>sbrk()</code>	P
FILE	<code>*freopen(const char *restrict filename, const char *restrict mode, FILE *restrict stream);</code>	<code>close()</code> <code>open()</code> <code>write()</code>	IE
int	<code>fscanf(FILE *restrict stream, const char *restrict format, ...);</code>	<code>close()</code> <code>lseek()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>fseek(FILE *stream, long offset, int whence);</code>	<code>lseek()</code> <code>write()</code>	IE
int	<code>fseeko(FILE *stream, off_t offset, int whence);</code> Under INTEGRITY, <code>off_t</code> is a 64-bit signed type. Elsewhere, it is <code>long</code> , so that <code>fseeko()</code> is the same as <code>fseek()</code> .	<code>lseek()</code> <code>write()</code>	IE

Function		Dependency	Thread Safety
int	<code>fsetpos(FILE *stream, const fpos_t *pos);</code>	<code>lseek()</code> <code>write()</code>	IE
long	<code>ftell(FILE *stream);</code>	<code>lseek()</code>	IE
off_t	<code>ftello(FILE *stream);</code> Under INTEGRITY, <code>off_t</code> is a 64-bit signed type. Elsewhere, it is <code>long</code> , so that <code>ftello()</code> is the same as <code>ftell()</code> .	<code>lseek()</code>	IE
int	<code>fwide(FILE *stream, int mode);</code>		N
int	<code>fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);</code>	<code>write()</code>	IE
size_t	<code>fwrite(const void *restrict ptr, size_t size, size_t nmemb, FILE *stream);</code>	<code>write()</code>	IE
int	<code>fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);</code>		IE
char	<code>*gcvt(double value, int ndig, char *buf);</code> Obsolete. Will be removed in the next release.		N
int	<code>getc(FILE *stream);</code>	<code>read()</code>	IE
int	<code>getc_unlocked(FILE *stream);</code>	<code>read()</code>	NE
int	<code>getchar(void);</code>	<code>read()</code>	IE
int	<code>getchar_unlocked(void);</code>	<code>read()</code>	NE
char	<code>*getenv(const char *name);</code>	<code>environ</code>	P
long	<code>getl(FILE *stream);</code> Obsolete. Will be removed in the next release.	<code>read()</code>	IE
char	<code>*gets(char *s);</code>	<code>read()</code>	IE
int	<code>getw(FILE *stream);</code>	<code>read()</code>	IE
wint_t	<code>getwchar(void);</code>	<code>read()</code>	IE
char	<code>*index(const char *str, const char ch);</code>		Y
int	<code>isalnum(int c);</code>		Y

Function	Dependency	Thread Safety
int isalpha(int c);		Y
int isblank(int c);		Y
int iscntrl(int c);		Y
int isdigit(int c);		Y
int isgraph(int c);		Y
int islower(int c);		Y
int isprint(int c);		Y
int ispunct(int c);		Y
int isspace(int c);		Y
int isupper(int c);		Y
int iswalnum(wint_t wc);		Y
int iswalpha(wint_t wc);		Y
int iswblank(wint_t wc);		Y
int iswcntrl(wint_t wc);		Y
int iswctype(wint_t wc, wctype_t desc);		Y
int iswdigit(wint_t wc);		Y
int iswgraph(wint_t wc);		Y
int iswlower(wint_t wc);		Y
int iswprint(wint_t wc);		Y
int iswpunct(wint_t wc);		Y
int iswspace(wint_t wc);		Y
int iswupper(wint_t wc);		Y
int iswxdigit(wint_t wc);		Y
int isxdigit(int c);		Y
long labs(long j);		Y
ldiv_t ldiv(long numer, long denom);		Y
long llabs(long long j);		Y
lldiv_t lldiv(long long numer, long long denom);		Y

Function	Dependency	Thread Safety
struct <code>*localeconv(void);</code> <code>lconv</code>		Y
void <code>longjmp (jmp_buf env, int val);</code>		Y
void <code>*malloc(size_t size);</code>	<code>sbrk()</code>	P
int <code>mblen(const char *s, size_t n);</code>		Y
size_t <code>mbrlen(const char *restrict s, size_t n, mbstate_t *restrict ps);</code>		E
size_t <code>mbrtowc(wchar_t *restrict pwc, const char *restrict s, size_t n, mbstate_t *restrict ps);</code>		E
int <code>mbsinit(const mbstate_t *ps);</code>		Y
size_t <code>mbsrtowcs(wchar_t *restrict dst, const char **restrict src, size_t len, mbstate_t *restrict ps);</code>		Y
size_t <code>mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);</code>		Y
int <code>mbtowc(wchar_t * restrict pwc, const char *restrict s, size_t n);</code>		Y
void <code>*memchr(const void *s, int c, size_t n);</code>		YK
int <code>memcmp(const void *s1, const void *s2, size_t n);</code>		YK
void <code>*memmove(void *s1, const void *s2, size_t n);</code>		YK
void <code>*memrchr(const void *s, int c, size_t n);</code>		Y
char <code>*mktemp(char *str);</code>	<code>getpid()</code>	P
time_t <code>mktime(struct tm *timeptr);</code>	<code>__gh_timezone()</code> <code>localtime()</code> <code>localtime_r()</code>	Y
void <code>perror(const char *s);</code>	<code>write()</code>	IE
int <code>printf(const char restrict *format, ...);</code>	<code>lseek()</code> <code>write()</code>	IE

Function		Dependency	Thread Safety
int	putc(int c, FILE *stream);	write()	IE
int	putc_unlocked(int c, FILE *stream);	write()	NE
int	putchar(int c);	write()	IE
int	putchar_unlocked(int c);	write()	NE
int	putenv(const char *c, int n);		PE
long	putl(long l, FILE *stream); Obsolete. Will be removed in the next release.	write()	IE
int	puts(const char *s);	write()	IE
int	putw(int w, FILE *stream);	write()	IE
wint_t	putwc(wchar_t wc, FILE *stream);	write()	IE
wint_t	putwchar(wchar_t wc);	write()	IE
void	qsort(void *base, size_t nmem, size_t size, int (*compar)(const void *, const void *));		Y
int	rand(void);		P
int	rand_r(long *seedSent);		Y
int	rand_r2(long seed[2])		Y
void	*realloc(void *ptr, size_t size);	sbrk()	P
int	remove(const char *filename);	unlink()	Y
void	rewind(FILE *stream);	lseek() write()	IE
char	*rindex(const char *str, const char ch);		Y
int	scanf(const char *restrict format, ...);	close() read() sbrk() write()	IE
void	setbuf(FILE *restrict stream, char *restrict buf);	close() write()	IE
int	setenv(const char *envname, const char *envval, int overwrite);		PE
int	setjmp (jmp_buf env);		Y
int	setlinebuf(FILE *stream);		N

Function		Dependency	Thread Safety
char	*setlocale(int category, const char *locale);		Y
int	setvbuf(FILE *restrict stream, char *restrict buf, int mode, size_t size);	close() write()	IE
int	snprintf(char *restrict s, size_t n, const char *restrict format, ...);		E
int	sprintf(char * restrict s, const char *restrict format, ...);		E
void	srand(unsigned int seed);		N
int	sscanf(const char *restrict s, const char *restrict format, ...);		Y
int	strcasecmp(const char *s1, const char *s2); Ignore the difference between uppercase and lowercase		Y
char	*strcat(char *restrict s1, const char *restrict s2);		Y
char	*strchr(const char *s, int c);		Y
int	strcmp(const char *s1, const char *s2);		Y
int	strcoll(const char *s1, const char *s2);		Y
char	*strcpy(char *restrict s1, const char *restrict s2);		Y
size_t	strcspn(const char *s1, const char *s2);		Y
char	*strdup(const char *s)		P
char	*strerror(int errnum);		Y
size_t	strftime(char *restrict s, size_t maxsize, const char *restrict format, const struct tm *restrict timeptr);	__gh_timezone()	Y
int	strindex(char *str, char *sub);		Y

Function		Dependency	Thread Safety
size_t	strlen(const char *s);		Y
int	strncasecmp(const char *s1, const char *s2, size_t n); Ignore the difference between uppercase and lowercase		Y
char	*strncat(char *restrict s1, const char *restrict s2, size_t n);		Y
int	strncmp(const char *s1, const char *s2, size_t n);		Y
char	*strncpy(char *restrict s1, const char *restrict s2, size_t n);		Y
char	*strndup(const char *s, size_t n)	sbrk()	P
size_t	strnlen(const char *s, size_t n)		Y
char	*strpbrk(const char *s1, const char *s2);		Y
char	*strrchr(const char *s, int c);		Y
int	strrindex(char *str, char *sub);		Y
char	*strsave(char *str);	sbrk()	P
size_t	strspn(const char *s1, const char *s2);		Y
char	*strstr(const char *s1, const char *s2);		Y
char	*strtok(char *restrict s1, const char *restrict s2);		T
char	*strtok_r(char *s1, const char *s2, char **ppLast);		Y
long	strtol(const char *restrict nptr, char **restrict endptr, int base);		E
unsigned long	strtoul(const char *restrict nptr, char **restrict endptr, int base);		E

Function		Dependency	Thread Safety
size_t	strxfrm(char *restrict s1, const char *restrict s2, size_t n);		Y
void	swab(char *from, char *to, int nbytes);		Y
int	swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict format, ...);		Y
int	swscanf(const wchar_t *restrict s, const wchar_t *restrict format, ...);		Y
FILE	*tmpfile(void);	close() open() unlink() write()	IE
char	*tmpnam(char *s);	access()	TPE
char	*tmpnam_r(char *s);	access()	PE
int	isascii(int ch); This function is implemented as a macro in ctype.h .		Y
int	toascii(int ch); This function is implemented as a macro in ctype.h .		Y
int	tolower(int c);		Y
int	toupper(int c);		Y
wint_t	towctrans(wint_t wc, wctrans_t desc);		Y
wint_t	towlower(wint_t wc);		Y
wint_t	towupper(wint_t wc);		Y
int	ungetc(int c, FILE *stream);	close() sbrk() write()	IE
wint_t	ungetwc(wint_t wc, FILE *stream);	close() sbrk() write()	IE
wint_t	ungetwc_unlocked(wint_t wc, FILE *stream);	close() sbrk() write()	NE
int	unsetenv(const char* envname)		P

Function		Dependency	Thread Safety
int	<code>vfprintf(FILE *restrict stream, const char *restrict format, va_list arg);</code>	<code>write()</code>	IE
int	<code>vfscanf(FILE *restrict stream, const char *restrict format, va_list arg);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>vfwprintf(FILE *restrict stream, const wchar_t *restrict format, va_list arg);</code>	<code>write()</code>	IE
int	<code>vfwscanf(FILE *restrict stream, const wchar_t *restrict format, va_list arg);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>vprintf(const char *restrict format, va_list arg);</code>	<code>write()</code>	IE
int	<code>vscanf(const char *restrict format, va_list arg);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
int	<code>vsnprintf(char *restrict s, size_t n, const char *restrict format, va_list arg);</code>		Y
int	<code>vsprintf(char *restrict s, const char *restrict format, va_list arg);</code>		Y
int	<code>vsscanf(const char *restrict s, const char *restrict format, va_list arg);</code>		Y
int	<code>vswprintf(wchar_t *restrict s, const wchar_t *restrict format, va_list arg);</code>		Y
int	<code>vswscanf(const wchar_t *restrict s, const wchar_t *restrict format, va_list arg);</code>		P
int	<code>vwprintf(const wchar_t *restrict format, va_list arg);</code>	<code>write()</code>	IE
int	<code>vwscanf(const wchar_t *restrict format, va_list arg);</code>	<code>close()</code> <code>read()</code> <code>sbrk()</code> <code>write()</code>	IE
size_t	<code>wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);</code>		Y

Function	Dependency	Thread Safety
wchar_t *wcscat(wchar_t *restrict s1, const wchar_t *restrict s2);		Y
wchar_t *wcschr(const wchar_t *s, wchar_t wc);		Y
int wcscmp(const wchar_t *s1, const wchar_t *s2);		Y
int wcscoll(const wchar_t *s1, const wchar_t *s2);		Y
wchar_t *wcscpy(wchar_t *restrict s1, const wchar_t *restrict s2);		Y
size_t wcsncpy(const wchar_t *s1, const wchar_t *s2);		Y
size_t wcsftime(wchar_t *restrict s, size_t maxsize, const wchar_t *restrict format, const struct tm *restrict timeptr);	__gh_timezone()	Y
size_t wcslen(const wchar_t *s);		Y
wchar_t *wcsncat(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);		Y
int wcsncmp(const wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);		Y
wchar_t *wcsncpy(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);		Y
wchar_t *wcpbrk(const wchar_t *s1, const wchar_t *s2);		Y
wchar_t *wcsrchr(const wchar_t *s, wchar_t wc);		Y
size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src, size_t len, mbstate_t *restrict ps);		Y
size_t wcsspncpy(const wchar_t *s1, const wchar_t *s2);		Y
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);		Y

Function	Dependency	Thread Safety
intmax_t wcstoimax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);		Y
wchar_t wcstok(wchar_t *restrict s1, const wchar_t *restrict s2, wchar_t **restrict ptr);		Y
long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);		Y
long wcstoll(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);		Y
size_t wcstombs(char * restrict s, const wchar_t *restrict pwcs, size_t n);		Y
unsigned wcstoul(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);		Y
unsigned wcstoull(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);		Y
uintmax_t wcstoumax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);		Y
size_t wcsxfrm(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);		Y
int wctob(wint_t wc);		Y
int wctomb(char *s, wchar_t wchar);		Y
wctrans_t wctrans(const char *property);		Y
wctype_t wctype(const char *property);		Y
wchar_t *wmemchr(const wchar_t *s, wchar_t wc, size_t n);		Y
int wmemcmp(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);		Y

Function	Dependency	Thread Safety
wchar_t *wmemcpy(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);		Y
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);		Y
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);		Y
int wprintf(const wchar_t *restrict format, ...);	write()	IE
int wscanf(const wchar_t *restrict format, ...);	close() read() sbrk() write()	IE



Note Memory allocated in `putenv()` is never freed. `unsetenv()` does not free any memory when it removes an entry from the environment.

libsys.a Function Implementation

This table lists the **libsys.a** low-level functions on which some **libansi.a** functions depend, and how the Green Hills tools implement each low-level function by default. If the default behavior is not adequate for your environment, you might need to re-implement the function.

Functions that Green Hills libraries implement as MULTI system calls communicate with your host machine. It is unlikely that you will need to change these functions when you connect to your target through MULTI. If you intend to continue using these functions without connecting to your target through MULTI, such as in a final product, you must re-implement these functions to suit your needs.

In the Nintendo environment, only `_Exit()`, `raise()`, `sbrk()`, and `write()` have been customized by Nintendo. In general, functions implemented in a MULTI system call will not work in a release build.

Function	Module	Implementation
<code>access()</code>	ind_stat.c	MULTI system call
<code>close()</code>	ind_io.c	MULTI system call
<code>creat()</code>	ind_io.c	MULTI system call
<code>_Exit()</code>	ind_exit.c	MULTI system call
<code>getpid()</code>	ind_gpid.c	Returns 17
<code>__gh_timezone()</code>	ind_tmzn.c	8 hours West of UTC/GMT
<code>localtime()</code>	ind_tmzn.c	US daylight savings rules
<code>localtime_r()</code>	ind_tmzn.c	US daylight savings rules
<code>lseek()</code>	ind_io.c	MULTI system call
<code>open()</code>	ind_io.c	MULTI system call
<code>raise()</code>	ind_sgnl.c	ISO C99 compliant
<code>read()</code>	ind_io.c	MULTI system call
<code>sbrk()</code>	ind_heap.c	Allocates from <code>.heap</code>
<code>time()</code>	ind_time.c	MULTI system call
<code>unlink()</code>	ind_io.c	MULTI system call
<code>write()</code>	ind_io.c	MULTI system call

You may also need to change `char **environ`, an object in **ind_crt1.c** that is implemented using MULTI argument passing by default.

libmath.a Functions

The following tables list functions in **libsys.a**, and are organized by the header file in which they are declared.

Functions from math.h in libmath.a

Function	Thread safety
int <code>__fpclassifyd(double x);</code>	Y
int <code>__fpclassifyf(double x);</code>	Y
int <code>__fpclassifyl(double x);</code>	Y
int <code>__isfinited(double x);</code>	Y
int <code>__isfinitef(double x);</code>	Y
int <code>__isfinitel(double x);</code>	Y
int <code>__isinfddouble x);</code>	Y
int <code>__isinff(double x);</code>	Y
int <code>__isinfl(double x);</code>	Y
int <code>__isnand(double x);</code>	Y
int <code>__isnanf(double x);</code>	Y
int <code>__isnanl(double x);</code>	Y
int <code>__isnormald(double x);</code>	Y
int <code>__isnormalf(double x);</code>	Y
int <code>__isnormall(double x);</code>	Y
int <code>__signbitd(double x);</code>	Y
int <code>__signbitf(double x);</code>	Y
int <code>__signbitl(double x);</code>	Y
double <code>acos(double x);</code>	E
float <code>acosf(float x);</code>	E
long double <code>acosl(long double x);</code>	E
double <code>acosh(double x);</code>	E
float <code>acoshf(float x);</code>	E
long double <code>acoshl(long double x);</code>	E
double <code>asin(double x);</code>	E

Function		Thread safety
float	<code>asinf(float x);</code>	E
long double	<code>asinl(long double x);</code>	E
double	<code>asinh(double x);</code>	Y
float	<code>asinhf(float x);</code>	Y
long double	<code>asinh1(long double x);</code>	Y
double	<code>atan(double x);</code>	Y
float	<code>atanf(float x);</code>	Y
long double	<code>atanl(long double x);</code>	Y
double	<code>atan2(double y, double x);</code>	Y
float	<code>atan2f(float y, float x);</code>	Y
long double	<code>atan2l(long double y, long double x);</code>	Y
double	<code>atanh(double x);</code>	E
float	<code>atanhf(float x);</code>	E
long double	<code>atanh1(long double x);</code>	E
double	<code>cbrt(double x);</code>	Y
float	<code>cbrtf(float x);</code>	Y
long double	<code>cbrtl(long double x);</code>	Y
double	<code>ceil(double x);</code>	Y
float	<code>ceilf(float x);</code>	Y
long double	<code>ceill(long double x);</code>	Y
double	<code>copysign(double x, double y);</code>	Y
float	<code>copysignf(float x, float y);</code>	Y
long double	<code>copysignl(long double x, long double y);</code>	Y
double	<code>cos(double x);</code>	Y
float	<code>cosf(float x);</code>	Y
long double	<code>cosl(long double x);</code>	Y
double	<code>cosh(double x);</code>	E
float	<code>coshf(float x);</code>	E
long double	<code>cosh1(long double x);</code>	E
double	<code>erf(double x);</code>	E
float	<code>erff(float x);</code>	E

Function		Thread safety
long double	<code>erfl(long double x);</code>	E
double	<code>erfc(double x);</code>	E
float	<code>erfcf(float x);</code>	E
long double	<code>erfcl(long double x);</code>	E
double	<code>exp(double x);</code>	E
float	<code>expf(float x);</code>	E
long double	<code>expl(long double x);</code>	E
double	<code>exp2(double x);</code>	E
float	<code>exp2f(float x);</code>	E
long double	<code>exp2l(long double x);</code>	E
double	<code>expm1(double x);</code>	E
float	<code>expm1f(float x);</code>	E
long double	<code>expm1l(long double x);</code>	E
double	<code>fabs(double x);</code>	Y
float	<code>fabsf(float x);</code>	Y
long double	<code>fabsl(long double x);</code>	Y
double	<code>fdim(double x, double y);</code>	E
float	<code>fdimf(float x, float y);</code>	E
long double	<code>fdiml(long double x, long double y);</code>	E
double	<code>floor(double x);</code>	Y
float	<code>floorf(float x);</code>	Y
long double	<code>floorl(long double x);</code>	Y
double	<code>fma(double x, double y, double z);</code>	Y
float	<code>fmaf(float x, float y, float z);</code>	Y
long double	<code>fmal(long double x, long double y, long double z);</code>	Y
double	<code>fmax(double x, double y);</code>	Y
float	<code>fmaxf(float x, float y);</code>	Y
long double	<code>fmaxl(long double x, long double y);</code>	Y
double	<code>fmin(double x, double y);</code>	Y
float	<code>fminf(float x, float y);</code>	Y

Function		Thread safety
long double	<code>fminl(long double x, long double y);</code>	Y
double	<code>fmod(double x, double y);</code>	E
float	<code>fmodf(float x, float y);</code>	E
long double	<code>fmodl(long double x, long double y);</code>	E
double	<code>frexp(double value, int *exp);</code>	Y
float	<code>frexpf(float value, int *exp);</code>	Y
long double	<code>frexpl(long double value, int *exp);</code>	Y
double	<code>gamma(double x);</code>	E
double	<code>hypot(double x, double y);</code>	E
float	<code>hypotf(float x, float y);</code>	E
long double	<code>hypotl(long double x, long double y);</code>	E
int	<code>ilogb(double x);</code>	Y
int	<code>ilogbf(float x);</code>	Y
int	<code>ilogbl(long double x);</code>	Y
int	<code>isfinite(double x);</code>	Y
int	<code>isinf(double x);</code>	Y
int	<code>isnan(double x);</code>	Y
int	<code>isnormal(double x);</code>	Y
double	<code>j0(double x);</code>	Y
double	<code>j1(double x);</code>	Y
double	<code>jn(int n, double x);</code>	Y
double	<code>ldexp(double x, int exp);</code>	E
float	<code>ldexpf(float x, int exp);</code>	E
long double	<code>ldexpl(long double x, int exp);</code>	E
double	<code>lgamma(double x);</code>	E
float	<code>lgammaf(float x);</code>	E
long double	<code>lgammal(long double x);</code>	E
double	<code>log(double x);</code>	E
float	<code>logf(float x);</code>	E
long double	<code>logl(long double x);</code>	E
double	<code>log10(double x);</code>	E

Function		Thread safety
float	<code>log10f(float x);</code>	E
long double	<code>log10l(long double x);</code>	E
double	<code>loglp(double x);</code>	E
float	<code>loglpf(float x);</code>	E
long double	<code>loglpl(long double x);</code>	E
double	<code>log2(double x);</code>	E
float	<code>log2f(float x);</code>	E
long double	<code>log2l(long double x);</code>	E
double	<code>logb(double x);</code>	E
float	<code>logbf(float x);</code>	E
long double	<code>logbl(long double x);</code>	E
double	<code>lrint(double x);</code>	E
float	<code>lrintf(float x);</code>	E
long double	<code>lrintl(long double x);</code>	E
long long	<code>llrint(double x);</code>	E
long long	<code>llrintf(float x);</code>	E
long long	<code>llrintl(long double x);</code>	E
long	<code>lround(double x);</code>	E
long	<code>lroundf(float x);</code>	E
long	<code>lroundl(long double x);</code>	E
long long	<code>llround(double x);</code>	E
long long	<code>llroundf(float x);</code>	E
long long	<code>llroundl(long double x);</code>	E
double	<code>modf(double value, double *iptr);</code>	Y
float	<code>modff(float value, float *iptr);</code>	Y
long double	<code>modfl(long double value, long double *iptr);</code>	Y
double	<code>nan(const char *tagp);</code>	Y
float	<code>nanf(const char *tagp);</code>	Y
long double	<code>nanl(const char *tagp);</code>	Y
double	<code>nearbyint(double x);</code>	Y

Function		Thread safety
float	<code>nearbyintf(float x);</code>	Y
long double	<code>nearbyintl(long double x);</code>	Y
double	<code>nextafter(double x, double y);</code>	E
float	<code>nextafterf(float x, float y);</code>	E
long double	<code>nextafterl(long double x, long double y);</code>	E
double	<code>nexttoward(double x, long double y);</code>	E
float	<code>nexttowardf(float x, long double y);</code>	E
long double	<code>nexttowardl(long double x, long double y);</code>	E
double	<code>pow(double x, double y);</code>	E
float	<code>powf(float x, float y);</code>	E
long double	<code>powl(long double x, long double y);</code>	E
double	<code>remainder(double x, double y);</code>	E
float	<code>remainderf(float x, float y);</code>	E
long double	<code>remainderl(long double x, long double y);</code>	E
double	<code>remquo(double x, double y, int *quo);</code>	E
float	<code>remquof(float x, float y, int *quo);</code>	E
long double	<code>remquol(long double x, long double y, int *quo);</code>	E
double	<code>rint(double x);</code>	Y
float	<code>rintf(float x);</code>	Y
long double	<code>rintl(long double x);</code>	Y
double	<code>round(double x);</code>	Y
float	<code>roundf(float x);</code>	Y
long double	<code>roundl(long double x);</code>	Y
double	<code>scalbn(double x, int n);</code>	E
float	<code>scalbnf(float x, int n);</code>	E
long double	<code>scalbnl(long double x, int n);</code>	E
double	<code>scalbln(double x, long int n);</code>	E
float	<code>scalblnf(float x, long int n);</code>	E

Function		Thread safety
long double	scalblnl(long double x, long int n);	E
double	sin(double x);	Y
float	sinf(float x);	Y
long double	sinl(long double x);	Y
double	sinh(double x);	E
float	sinhf(float x);	E
long double	sinhl(long double x);	E
double	sqrt(double x);	E
float	sqrtf(float x);	E
long double	sqrtl(long double x);	E
double	tan(double x);	E
float	tanf(float x);	E
long double	tanl(long double x);	E
double	tanh(double x);	E
float	tanhf(float x);	E
long double	tanh1(long double x);	E
double	tgamma(double x);	E
float	tgammaf(float x);	E
long double	tgammal(long double x);	E
double	trunc(double x);	Y
float	truncf(float x);	Y
long double	truncl(long double x);	Y
double	y0(double x);	E
double	y1(double x);	E
double	yn(int n, double x);	E

Functions from fenv.h in libmath.a

Function		Thread safety
void	<code>feclearexcept(int excepts);</code>	Y
void	<code>fegetenv(fenv_t *envp);</code>	Y
void	<code>fegetexceptflag(fexcept_t *flagp, int excepts);</code>	Y
int	<code>fegetround(void);</code>	Y
int	<code>feholdexcept(fenv_t *envp);</code>	Y
void	<code>feraiseexcept(int excepts);</code>	Y
void	<code>fesetenv(const fenv_t *envp);</code>	Y
void	<code>fesetexceptflag(const fexcept_t *flagp, int excepts);</code>	Y
int	<code>fesetround(int round);</code>	Y
int	<code>fetestexcept(int excepts);</code>	Y
void	<code>feupdateenv(const fenv_t *envp);</code>	Y

Functions from complex.h in libmath.a

Function		Thread safety
double complex	<code>cabs(double complex z);</code>	E
float complex	<code>cabsf(float complex z);</code>	E
long double complex	<code>cabsl(long double complex z);</code>	E
double complex	<code>cacos(double complex z);</code>	E
float complex	<code>cacosf(float complex z);</code>	E
long double complex	<code>cacosl(long double complex z);</code>	E
double complex	<code>cacosh(double complex z);</code>	E

Function		Thread safety
float complex	<code>cacoshf(float complex z);</code>	E
long double complex	<code>cacoshl(long double complex z);</code>	E
double complex	<code>casin(double complex z);</code>	E
float complex	<code>casinf(float complex z);</code>	E
long double complex	<code>casinl(long double complex z);</code>	E
double complex	<code>casinh(double complex z);</code>	E
float complex	<code>casinhf(float complex z);</code>	E
long double complex	<code>casinhl(long double complex z);</code>	E
double complex	<code>carg(double complex z);</code>	E
float complex	<code>cargf(float complex z);</code>	E
long double complex	<code>cargl(long double complex z);</code>	E
double complex	<code>catan(double complex z);</code>	E
float complex	<code>catanf(float complex z);</code>	E
long double complex	<code>catanl(long double complex z);</code>	E
double complex	<code>catanh(double complex z);</code>	E
float complex	<code>catanhf(float complex z);</code>	E
long double complex	<code>catanhl(long double complex z);</code>	E
double complex	<code>ccos(double complex z);</code>	E

Function		Thread safety
float complex	<code>ccosf(float complex z);</code>	E
long double complex	<code>ccosl(long double complex z);</code>	E
double complex	<code>ccosh(double complex z);</code>	E
float complex	<code>ccoshf(float complex z);</code>	E
long double complex	<code>ccoshl(long double complex z);</code>	E
double complex	<code>cexp(double complex z);</code>	E
float complex	<code>cexpf(float complex z);</code>	E
long double complex	<code>cexpl(long double complex z);</code>	E
double complex	<code>cimag(double complex z);</code>	E
float complex	<code>cimagf(float complex z);</code>	E
long double complex	<code>cimagl(long double complex z);</code>	E
double complex	<code>clog(double complex z);</code>	E
float complex	<code>clogf(float complex z);</code>	E
long double complex	<code>clogl(long double complex z);</code>	E
double complex	<code>conj(double complex z);</code>	E
float complex	<code>conjf(float complex z);</code>	E
long double complex	<code>conjl(long double complex z);</code>	E
double complex	<code>cpow(double complex x, double complex y);</code>	E

Function		Thread safety
float complex	<code>cpowf(float complex x, float complex y);</code>	E
long double complex	<code>cpowl(long double complex x, long double complex y)</code>	E
double complex	<code>cproj(double complex z);</code>	E
float complex	<code>cprojf(float complex z);</code>	E
long double complex	<code>cprojl(long double complex z);</code>	E
double complex	<code>creal(double complex z);</code>	E
float complex	<code>crealf(float complex z);</code>	E
long double complex	<code>creall(long double complex z);</code>	E
double complex	<code>csin(double complex z);</code>	E
float complex	<code>csinf(float complex z);</code>	E
long double complex	<code>csinl(long double complex z);</code>	E
double complex	<code>csinh(double complex z);</code>	E
float complex	<code>csinhf(float complex z);</code>	E
long double complex	<code>csinhl(long double complex z);</code>	E
double complex	<code>csqrt(double complex z);</code>	E
float complex	<code>csqrtf(float complex z);</code>	E
long double complex	<code>csqrtl(long double complex z);</code>	E
double complex	<code>ctan(double complex z);</code>	E

Function		Thread safety
float complex	<code>ctanf(float complex z);</code>	E
long double complex	<code>ctanl(long double complex z);</code>	E
double complex	<code>ctanh(double complex z);</code>	E
float complex	<code>ctanhf(float complex z);</code>	E
long double complex	<code>ctanhl(long double complex z);</code>	E

Functions from `stdlib.h` in `libmath.a`

Function		Thread safety
double	<code>atof(const char *nptr);</code>	E
double	<code>strtod(const char *restrict nptr, char **restrict endptr);</code>	E
float	<code>strtof(const char *restrict nptr, char **restrict endptr);</code>	E
long double	<code>strtold(const char *restrict nptr, char **restrict endptr);</code>	E

Functions from `wchar.h` in `libmath.a`

Function		Thread safety
double	<code>wctod(const wchar_t *restrict nptr, wchar_t **restrict endptr);</code>	E
long double	<code>wctold(const wchar_t *restrict nptr, wchar_t **restrict endptr);</code>	E
float	<code>wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);</code>	

libind.a Functions

Function		Thread safety
int	<code>_rterr(int num, int linenum, char *str, void *ptr, void **trace, int depth, size_t len)</code>	Y
int	<code>_rterr2(int num, int linenum, char *str)</code>	Y
intmax_t	<code>imaxabs(intmax_t, intmax_t)</code>	Y
imaxdiv_t	<code>imaxdiv(intmax_t, intmax_t)</code>	Y
intmax_t	<code>strtoimax(const char * restrict, char **restrict, int)</code>	E
long long	<code>strtoll(const char * restrict, const char **restrict, int)</code>	E
unsigned long long	<code>strtoull(const char * restrict, const char **restrict, int)</code>	E
uintmax_t	<code>strtoumax(const char * restrict, char **restrict, int)</code>	E



Note The functions `_rterr()` and `_rterr2()` are used in run-time error checking. For more information about run-time error checking, see “**Run-Time Error Checks**” on page 188.

libstartup.a Functions

The following table lists the standard C library functions in **libstartup.a**.

Function		Thread safety
void	<code>*memcpy(void * restrict d, const void * restrict s, size_t n)</code>	YK
void	<code>*memset(void *s, int c, size_t n);</code>	YK

Customizing the Run-Time Environment Libraries and Object Modules

The following sections contain information about the libraries and modules that make up the Green Hills run-time environment, and explain how to create a project that allows you to customize them.

Creating a Project with a Customizable Run-Time Environment

While the standard Green Hills run-time environment is sufficient for most users, you may need to customize some of its underlying routines to suit your needs. To create a project that allows you to customize the run-time environment:

1. If you do not have an existing project, click the  button in the Project Manager to open the **Project Wizard**, and select the appropriate settings until you reach the **Link Options** screen.

If you have an existing project, right-click a program in that project and select **Configure**.

2. On the screen that allows you to specify your **Program Layout**, select one or more of the **Libraries** check boxes (see also “Settings for Stand-Alone Programs” on page 31). If you set your **Program Layout** to **Link to ROM and Execute out of RAM**, some startup code runs out of ROM (see “Assigning Program Sections to ROM and RAM” on page 132).

The **Project Wizard** adds the following files your target resources project (**tgt/resources.gpj**), depending on the check boxes you selected:

- **libstartup/startup.gpj** — A project that contains the following startup code:
 - **crt0.ppc** — see “Startup Module crt0.o” on page 753.
 - **libstartup.gpj** — see “Low-Level Startup Library libstartup.a” on page 755.
- **libsys/libsys.gpj** — see “Low-Level System Library libsys.a” on page 756.
- **libboardinit/libboardinit.gpj** — see “Board Initialization Library libboardinit.a” on page 760. This library is not available for all targets.

You can customize the source to the startup modules as necessary and rebuild them separately, or as part of your project. Be careful when making any changes

to the build options for these projects, because some of them are required for proper operation. For example, **libstartup/startup.gpj** and **libsys/libsys.gpj** define the preprocessor symbol `EMBEDDED` when compiling. This symbol is required for some modules in the libraries. For more information about building and setting build options, see Chapter 2, “The MULTI Project Manager”.



Note The interface to these modules might change between releases of the product. If you customize your run-time environment and later upgrade to a new release, you may need to propagate your customizations to the run-time environment shipped with that new releases. You may not be able to combine the run-time environment from a given release with other libraries in a later release.

Working With Makefiles

If you are using makefiles to invoke the compiler driver and want to customize your run-time environment:

1. Create your project with the **Project Wizard** using the steps provided in “Creating a Project with a Customizable Run-Time Environment” on page 751.
2. Modify each run-time environment file as needed.
3. Use your makefile to link with the modified files.

Optimizing Startup Code

If you want to build faster, smaller startup code with reduced functionality, define the `MINIMAL_STARTUP` preprocessor symbol in the **libstartup/libstartup.gpj** and **libsys/libsys.gpj** projects. This startup code may be useful for very small and simple test applications that run out of RAM. The following list contains some of the features that are not supported in this library mode:

- ROM to RAM copy
- multithreading
- manual profiling

This library mode is provided as is. It is intended for experienced users who understand its limited feature set.

Startup Module **crt0.o**

crt0.o is the default startup module linked with your program. The source code for **crt0.o** is contained in **libstartup/startup.gpj/crt0.ppc**. This file contains the `_start()` function, which is the default entry point for the program. If you set your **Program Layout** to Link to ROM and Execute out of RAM, the initialization code in this module is placed in and executes out of the uncompressed `.ROM.boottext` section.

crt0.ppc

crt0.ppc is a small target-specific assembly language file containing code to set up a C program environment before calling into a C function. It executes as follows:

1. On startup, the code uses a system call to determine whether a debug server is controlling execution of the program (as opposed to the program running stand-alone, without being connected to a debug server).
2. If the system call is successful, **crt0.ppc** assumes that the debug server has already performed some register initialization (such as the stack pointer).

If the system call is not successful, **crt0.ppc** initializes additional registers, including the stack pointer. Consult the code to determine which other registers are manipulated, because this varies across architectures.

3. After register initialization, `_start()` calls the function `__ghs_ind crt0()`, the target-independent startup routine located in **libstartup.a**.

Customizing **crt0.ppc**

If you create your own startup code, you must initialize fixed registers (defined by your processor's ABI) to point to the correct memory regions. The register most commonly requiring initialization is the stack pointer, because a valid stack is generally required before a high-level language function can be called.

When initializing the stack pointer, the following information may be helpful:

- The `.stack` section defined in your linker directives (**.ld**) file specifies the location and size of the run-time stack.

For example, the following excerpt from a linker directives file specifies that the stack starts on the first 16-byte aligned address following the `.heap` section in memory. The stack is configured to be 0x80000 bytes:

```
.heap align(16) pad(0x100000) :  
.stack align(16) pad(0x80000) :
```

- `__ghsbegin_stack` and `__ghsend_stack` specify the beginning and the end of the `.stack` section. For more information about special symbols of this format, see “Beginning, End, and Size of Section Symbols” on page 463.
- The stack pointer should be initialized to the end of this section, because it grows downward. To check if the stack is exhausted at run time, compare the stack pointer to `__ghsbegin_stack`.

Low-Level Startup Library **libstartup.a**

The **libstartup.a** library contains code that copies sections of a program's text and initialized data from ROM to RAM. The code is contained in **libstartup.gpj**.

If you set your **Program Layout** to **Link to ROM and Execute out of RAM**, the initialization code in this library is placed in and executes out of the uncompressed `.ROM.boottext` section.

ind crt0.c

This module contains startup code in the function `__ghs_ind crt0()`. This function:

1. clears uninitialized sections such as `.bss`.
2. copies initialized sections from their locations in ROM to their final locations in RAM, as specified in your linker directives file.
3. decompresses compressed initialized sections from their locations in ROM and copies them to their final location in RAM. For more information, see “Copying a Section from ROM to RAM at Startup” on page 456.
4. calls the library initialization routine `__ghs_ind crt1()` in the `.text` section through a function pointer. The application then executes code in the `.text` section. For more information, see “ind crt1.c” on page 757.

ind_mcopy.ppc, ind_mset.ppc

These modules contain the C run-time library functions `memcpy()` and `memset()`, which are used during initialization to copy and clear data sections.

ind_reset.ppc

This module contains low-level initialization routines. You may need to modify it for your target.

ind_uzip.c

This module contains code that decompresses program sections when copying them from ROM to RAM. It is pulled in by the linker if any CROM sections are contained in the program.

ind_initmem.c

This module is not used on the PowerPC architecture.

Low-Level System Library libsys.a

The **libsys.a** library contains code that performs run-time library initialization, profiling, and system calls. The source code is contained in *install_dir/src/libsys*.

Several **libsys.a** I/O functions perform system calls on the host when you are connected to your target with the MULTI Debugger. For more information, see “libsys.a Function Implementation” on page 737.

ind_alloc.c

This module contains functions that allocate memory of the given type and alignment from the heap. These functions call `sbrk()` (in **ind_heap.c**) to allocate the memory, but also provide alignment guarantees.

ind_call.ppc, ind_dots.ppc

These modules implement low-level system call capability. The routine `__ghs_syscall()` is called by various system call routines, such as `open()`, `close()`, `read()`, and `write()`. The `__ghs_syscall()` routine transfers control to the start address of the `.syscall` section.

Debug servers can set a breakpoint at the start of the `.syscall` section to emulate system calls on the host. When the breakpoint is hit, the debug server knows that a system call occurred. The arguments are retrieved and the system call is emulated on the host.

ind_crt1.c

This module contains the first C function called by the standard libraries after memory and static and global data have been initialized. It initializes the ANSI C and other run-time libraries before jumping to the application entry point (`main`).

ind_errn.c

This module handles reads and writes to `errno` and can be customized to maintain a per-thread `errno` value. See also **ind_thrd.c**.

ind_gpid.c

This module returns the current process number. This is used in `mktemp()` to add uniqueness to the generated filenames.

ind_lock.c

This module implements the global locking mechanism needed for thread-safe libraries.

ind_manprf.c

This module contains code to perform stochastic target-based timing profiling when enabled by the **-timer_profile** option (see “Target-Based Timing Profiling” on page 120).

ind_stackcheck.c

This module contains code to perform stack checking when enabled by the **-stack_check** option.

ind_targ1.c

This is an empty file for the PowerPC architecture. It is deprecated and may be removed in future versions of **MULTI**.

ind_thrd.c

This module implements the file-specific locking mechanism and optional per-thread run-time library data structures. In addition, this module provides the following functions to allow saving and restoring arbitrary state across `setjmp()` and `longjmp()` calls:

```
int __ghs_SaveSignalContext(jmp_buf jmpbuf)
void __ghs_RestoreSignalContext(jmp_buf jmpbuf)
```

ind_mcnt.ppc, ind_gcnt.ppc, ind_bcnc.c, ind_mprf.c, ind_gprf.c

These modules provide profiling support. The routine `__ghs_mcount()` is used for call count profiling and `__ghs_gcount()` is used for call graph profiling. The module **ind_bcnc.c** contains the profiling routine `__ghs_bcount()`, which handles calls that are emitted by the compiler to implement code coverage analysis.

ind_heap.c

This module contains the dynamic memory allocation routine `sbrk()`. The `.heap` section in the linker directives file specifies the location, size, alignment, and initialization, if any, of the heap. For example:

```
.heap align(16) pad(0x100000) :
.stack align(16) pad(0x80000) :
```

This specifies that the heap starts on the first 16-byte aligned address following the previous section in memory and is 0x100000 bytes in size. The `.stack` section then follows the heap area. The `pad` directive instructs the linker that the heap is uninitialized on startup. This enables you to place the run-time heap anywhere in memory; the run-time library automatically allocates memory where you place this section.

Specifying a size for the heap helps ensure that your program does not use more heap memory than expected. Any attempt to allocate more memory than specified in `.heap` causes `sbrk()` to fail and return `(void*) -1`.

ind_io.c

This module contains default implementations of system routines most likely needed for a Linux/Solaris-like implementation, including:

```
int open(const char *filename, int mode, ...);
int creat(const char *filename, int prot);
int close(int fno);
int read(int fno, void *buf, int size);
int write(int fno, const void *buf, int size);
int unlink(char *name);
```

These system calls enable the program to open, read, write, and close files.

The calls in **ind_io.c** perform these operations on the host using the system call interface routine `__ghs_syscall()`, described in “ind_call.ppc, ind_dots.ppc” on page 756.

ind_iob.c

This module contains the default file buffer, `_iob`, used by the C library’s **stdio.h** interface. The user may want to modify this to reduce or increase the number of files that may be opened through this interface. The user may also modify the default buffering by modifying `__gh_iob_init()`.

ind_exit.c

This module contains the following routines:

Routine	Description
<code>__ghs_at_exit(struct __GHS_AT_EXIT*)</code>	Registers functions that need to be called upon <code>_Exit()</code> (similar to the ANSI <code>atexit()</code>).
<code>void _Exit(int)</code>	Calls clean-up routines registered by the program via <code>__ghs_at_exit()</code> and allows the program to terminate cleanly via the <code>__ghs_syscall(SYSCALL_EXIT, code)</code> system call. Clean-up routines registered by <code>atexit()</code> and C++ global destructors are handled separately in <code>exit()</code>, which finishes by calling <code>_Exit()</code>.

Other Low-Level Functions

The following files contain a basic set of Linux/Solaris-like operating system routines. Each routine is commented to show the function it performs and the values it returns. These routines allow the linker to resolve an operating system's symbols, referenced by the Green Hills run-time libraries.

Source File	Routine(s)
ind_gmtm.c	<code>struct tm *gmtime(const time_t *timer);</code>
ind_renm.c	<code>int rename(const char *old, const char *new);</code>
ind_sgnl.c	<code>int raise(int sig);</code> <code>void (*signal(int sig, void (*func)(int)))(int);</code>
ind_stat.c	<code>int access(char *name, int mode);</code>
ind_syst.c	<code>int system(const char *string);</code>
ind_time.c	<code>time_t time(time_t *tptr);</code> <code>int times (struct tms *buffer);</code>
ind_tmzn.c	<code>struct tm *localtime(const time_t *timer);</code> <code>void tzset(void);</code> <code>int __gh_timezone(void);</code>
ind_trnc.c	<code>int truncate(const char* path, long length);</code>

Board Initialization Library **libboardinit.a**

If you choose to customize the **Board Initialization** library in the **Project Wizard** (see “Creating a Project with a Customizable Run-Time Environment” on page 751), it adds **libboardinit/libboardinit.gpj** to your Top Project. This project builds **libboardinit.a**, which varies by target and might contain some or all of the following features.

- ROM-based program support. This includes linker directives files and code to initialize the memory, chip selects, TLBs, and caches to allow the program to execute starting from reset.
- Cache synchronization support. This consists of the routine `void __ghs_board_cache_sync(void* s, size_t n)`, which is called after each ROM-to-RAM copy of a section containing program text. This routine flushes the data cache and invalidates the instruction cache contents starting at address *s* and continuing for *n* bytes.

- Serial port support. This might include routines to transmit and receive characters, as well as optionally including an example of an interrupt service routine to handle serial port interrupts.
- I/O library overrides. This includes code to connect the C/C++ library's `stdin` input stream and `stdout` output stream to the serial port. This code defines alternate versions of the `read()` and `write()` system routines that are otherwise defined by **ind_io.c** in the low-level system library **libsys.a**.
- Timer support. This consists of code that is called when target-based timer profiling is enabled. For more information about this feature, see “Target-Based Timing Profiling” on page 120.

The memory initialization code can be placed either at the reset vector location or in a routine `__ghs_board_memory_init`, which is called from the Startup Module **crt0.o** if it exists. After this routine returns, the memory defined by the linker directives file must be accessible to the program. This routine should be written in assembly because C code might inadvertently rely on stack memory being accessible.

The remaining device initialization code for the serial port, the interrupt controller, the I/O library overrides, and the timer setup code is placed in the routine `__ghs_board_devices_init`. `__ghs_board_devices_init` can be written in C because memory will be initialized, although not all of the C library is usable.

If the file **board_init.txt** was automatically included in your new project, It contains further information about debugging stand-alone and other programs on your board.

Features that Depend on the Default Green Hills Startup Code

This section provides a partial list of features and options that depend on the Green Hills startup code. If you customize this code, the following features might not work as expected:

Feature	Startup Code Requirement
<code>atexit()</code>	Processing in the Green Hills <code>_exit()</code> function.
C++ destructors	The Green Hills <code>exit()</code> function.
Multithreading	Might require <code>__gh_lock_init()</code> to be called.

Feature	Startup Code Requirement
PIC and PID	Might require base registers to be set up.
stdin, stdout, and stderr	<code>__gh_iob_init()</code> must be called.

The following options also depend on Green Hills startup code:

Option	Startup Code Requirement
-a -p -pg	The Green Hills <code>exit()</code> or <code>_exit()</code> function's dump profiling data. To learn how to dump profiling data during a program's execution, see "Manually Dumping Profiling Data" in Chapter 15, "Collecting and Viewing Profiling Data" in the <i>MULTI: Debugging</i> book.
-additional_sda_reg	Requires Green Hills startup code.
-globalreg	Certain variables (such as those declared with <code>register int glob</code>) are not initialized to the value 0 unless you use Green Hills startup code.



Note This is not a complete list of features that depend on Green Hills startup code. Because customizing this code may cause certain features to exhibit unexpected behavior, do so only when necessary.

Mixing Languages and Writing Portable Code

Chapter 18

This Chapter Contains:

- Mixing Languages
- Writing Portable Code

Mixing Languages

Green Hills compilers can combine C and C++ routines within the same executable files, subject to certain constraints. This section provides some basic information about using Green Hills compilers to create a mixed language executable and describes how the compiler driver builds mixed language executables. Additional language-specific details are provided in the subsequent sections.

All Green Hills compiler drivers are compatible with each other. This permits a C driver to compile a C++ module, and a C++ driver to compile a C module. The driver uses the input filename extension to determine the correct language, rather than assuming that the name of the driver determines the source code language.

Though compatible during compilation, the various drivers differ during the link phase. To link an application, the driver must determine all of the languages in use in order to know which libraries to include. The driver assumes that every application has modules written in C and assembly language, and that there is at least one module written in the driver's default language. If the command line includes source files written in other languages (as indicated by the file extension), then the driver recognizes that those languages exist in the application as well.

Consequently, mixing any one language with C is easy because the driver always assumes C is in use. To ensure the correct linkage when mixing a language with C, use the driver for the language other than C for linking the application.

Initialization of Libraries

A multiple language application may need to perform input and output in more than one language. With a little care to avoid conflicts between languages, this is fully supported. If input and output will always be performed on different files by each language, then the main program in a single language application automatically handles the initialization and deinitialization of each language's run-time routines. Therefore, if the application will only perform I/O in one language other than C, then it is easy to write the main program for the application in that language. For more complex requirements, write a main program in C to perform the initialization and deinitialization of the library run-time routines. Examples of main programs in C are given in the next section.

Examples of main() Programs

All language environments automatically perform initializations at the start of the execution and cleanup at the termination. The initializations and cleanup vary depending on the language. Examples of initializations and cleanup include command line argument initialization, standard input/output configuration, and static object creation/destruction. Since the C language initializations and cleanup differ from that of the other languages, you must often insert code inside the `main()` function to ensure proper configuration.

C main() Program for C++

If a program uses both C and C++, and the `main()` function is written in C, insert a call to `_main()` as the first executing statement and a call to `exit(0)` as the last executing statement (see example below). You must do this to ensure that static constructors and destructors are properly configured. For more details on using C++ in C programs refer to "Using C++ in C Programs" on page 767.

```
void main(void)
{
    _main(); /* must be first executable line */
    /* rest of main goes here */
    exit(0); /* must be last executable line */
}
```

Performing I/O on a Single File in Multiple Languages

Some applications benefit from performing input and output on a single file or device from more than one language; one example of this is pre-opened files. In C, these are `stdin`, `stdout`, and `stderr`. In C++, they are the same as C or, alternatively, `istreams` `cin`, `cout`, or `cerr` may be used.

All languages have full access to these pre-opened files, and input and output can easily be mixed between the languages on these files. However, for the best results, a complete input or output operation is done in a single language. In C, any call to a library function which performs input or output is a complete operation. If this rule is followed, all data will be output correctly and in the intended sequence. The C library routine `fflush()` flushes the buffer of the pre-opened files in all languages. In addition, to flush one of C++ `istreams`, use the notation `file << flush`. For example:

```
std::cout << std::flush;
```

Performing input and output on a single file which is not preopened is more difficult. It is possible to open the file once in each language and perform input and output independently in each language. In many cases this would be unacceptable, particularly when working with a device rather than a simple file.

C Routines and Header Files In C++

The C++ language allows much use of existing C code. Therefore, it is fairly straightforward to call functions written in ANSI C from C++. The syntax of the two languages is very similar and the use of header files has been continued in C++.

By default in C++, the names of functions are encoded or mangled, whereas in C, the names of functions are unchanged. C++ provides the `extern` specifier to identify non-C++ functions so their names will not be mangled. Therefore, this specifier allows ANSI declarations for any C functions to be included and then linked with the compiled C object code.

To specify a C declaration:

- Specify or declare functions individually. The following example specifies that two functions with `extern "C"` linkage are to be declared.

```
extern "C" {  
    int fclose(FILE *);  
    FILE *fopen(const char *, const char *);  
}
```

We recommend that you do not use a function declaration with an empty parameter list. This means different things in C and C++ (see “Function Prototyping in C vs. C++” on page 768). For compatibility in both languages, specify the parameters, or use a parameter list consisting solely of the keyword `void` if there are no parameters. For example:

```
int func1(int a, short b, const int *c); /* GOOD */  
void func2(void);                       /* GOOD */  
int func3();                             /* BAD */
```

- If code contains both C and C++, then the `extern` constructs can be placed within `#ifdef __cplusplus` statements. This practice is common within header files. For example,

```
#ifdef __cplusplus  
extern "C" {  
#endif  
void assert(int);  
void _assert(const char *,const int,const char *);  
#ifdef __cplusplus  
}  
#endif
```

Using C++ in C Programs

To call C++ from C, declare an `extern "C"` interface for the C++ code. Otherwise, the C code must manually perform some of the tasks that the C++ compiler performs automatically. You must know the following internal mechanics and details of the C++ implementation:

- The C++ compiler encodes or mangles function and class member names. Any C++ function or class members called from a C program must be referenced by the encoded or mangled names.
- The manner in which a C++ compiler handles member functions must be known. All member functions (except static member functions) have the special object member pointer `this` inserted automatically as the first argument in the parameter list. A C programmer must manually add the argument `this` when calling any member functions from C.

- Handling constructors and destructors for static objects requires special processing. On most systems, the `main()` function has a special function call to `_main()` inserted to ensure that all static constructor/destructor calls are made properly. If `main()` is not in a C++ module, then the C programmer must manually include calls to `_main()` in the C main module. The `_main()` code is contained in the C++ library and therefore must be linked into the final executable.



Note When using INTEGRITY and creating a C task, `_main` is called before `main()` so that if `main()` is in C code but there are C++ static constructors, the call is still constructed.

- Virtual functions are also handled automatically by a C++ compiler, but involve additional coding to access or use them from a C environment.

Function Prototyping in C vs. C++

In ANSI C and C++, header files provide prototypes for library functions which enforce a standard interface between the calling program and the called function.

Function prototyping requires that the function declaration include the function return type and the number and type of the arguments. When a prototype is available for a function, the compiler is able to perform argument checking and coercion on calls to that function. If a prototype is not available for a function when it is called, ANSI C will behave like K&R C. The return type of the function is assumed to be `int`, and actual arguments will be promoted to `int`, `long`, or `double` types, or pointers as appropriate. In C++, however, it is an error to call a function which has not been declared with a prototype.

Another important difference between ANSI C and C++ is that a non-prototype declaration of a function, such as:

```
char *function_name();
```

has no effect on the number and type of the arguments in ANSI C. In C++ it is understood as:

```
char *function_name(void)
```

which means that the function has no arguments at all. If the function declaration occurs within the scope of an `extern "C"` declaration, the function has

non-C++ linkage, and therefore cannot be overloaded. This means that if a traditional K&R style declaration of a function appears in a header file, and the `#include` directive which accesses that header file is enclosed in `extern "C" { }`, then it will be impossible to redeclare that function with arguments.

Mixing C and Assembly Language

The *PowerPC Embedded Application Binary Interface (PowerPC EABI)* specifies rules for calling subroutines and passing parameters. These rules are followed by the Green Hills compiler family. Assembly language programs that follow the same conventions for passing and returning parameters and for using volatile and non-volatile registers will be able to call and be called from C programs. In addition, assembly language programmers may want to access C variables and data structures or declare variables that are accessible from C programs.

The PowerPC Architecture documents the assembly language instruction and register mnemonics used in Green Hills assembly language. Information about specific PowerPC processors is available in *PowerPC 601*, Motorola Inc.; *Motorola MPC500 Family RCPU Reference Manual*; *SIU Reference Manual*; and similar sources.

Identifiers

Variable names are case-sensitive in both C and assembly language. All characters in an identifier are always significant. There is no limit on the maximum number of characters in an identifier.

According to the *PowerPC EABI*, the convention for naming variables is the following:

Global and external variables are named in assembly language modules exactly as they are named in C modules. Case is significant. For example, the following C variable:

```
int x;
```

could be accessed by an assembly language statement:

```
y:    .long x
```

This would initialize variable `y` to contain the address of C variable `x`.

Static variables are not visible outside the C source file, but naming issues arise when `asm` statements put assembly language code into C modules. File scope static variables are named the same in C, just as with global variables. Static variables that are local to functions are given a unique name generated internally by the compiler.

Calling Subroutines

Function entry points are also named in assembly language exactly as C source files. A C function:

```
void sort () {}
```

would be called by the assembly language statement:

```
bl sort
```

Similarly, a C routine calling an assembly language routine:

```
asmsort (1,2,3);
```

would require an assembly language definition, such as:

```
.globl asmsort
asmsort:
    # code here
```

Accessing Variables

In addition to the rules for naming, you should declare all assembly language symbols that need to be visible to other modules in `.globl` statements. Symbols defined externally in other modules should be declared in `.extern` statements.

In general, variables may either be local stack variables or file scope (sometimes called global scope) variables that take up space in the load map. The Green Hills PowerPC assembler does not provide facilities for named stack frame variables, other than the `=` feature to give symbolic names to constants. For example, a variable `X` referenced at offset 12 off the stack pointer might be handled by:


```
X = 12
    lwz r0, X(sp)    # Load X
```

Accessing C Arrays

C language arrays may be shared between assembly language modules using similar techniques. Declare the variables in `.globl` statements if they are defined in the assembly language module. They may be referenced from other modules and in `.extern` statements if they are defined in external modules and referenced locally. An array can be treated as a variable that is larger in size than a scalar variable. Indexing operations are performed by calculating an offset from the address of the variable appropriate for the index.

For example, the following C language variable definition:

```
int x[100];
```

is typically accessed by assembly language code:

```
.extern x
lis r3, %hi(x)      # Load address of x into r3
ori r3, r3, %lo(x)  # Complete the load
lwz r4, 40(r3)      # Load x[10] into r4
stw r4, 4(r3)       # Store r4 to x[1]
```

An equivalent assembly language definition of `x`, suitable for access from other modules, is:

```
.globl x
.comm x, 400        # Allocate 400 byte zero-initialized var
```

Accessing C Structures

A slightly more elaborate mechanism is provided by the assembler to define a group of symbols at appropriate offsets based on data space sizes. This mechanism uses the `.dsect` directive to define symbol names suitable for stack variables and declare symbolic offsets suitable for accessing C structure members. You can use this at the beginning of a block of code that ends with a `.end` directive. For example:

```
.dsect
    .long 0,0          # unused stack area
args:
    .long 0,0,0,0      # output argument area
x:   .long 0           # integer variable
f:   .double 0         # double precision
pt:  .space 16         # C structure of four ints
.end
```

The above code gives numeric values to the symbols appearing as labels within the `.dsect` directive. The legal directives used within `.dsect` are memory-allocating directives, such as `.space`, `.byte`, and `.long`. The values after `.byte` and `.long` are not used; only the number of such values is significant in determining the interval between symbols defined as labels within the `.dsect`. In the example above, `args` is given the value eight, which is the offset from the beginning of the `.dsect` block; `x` is given the value 24; `f` is given the value 28; and `pt` is given the value 36. The same effect can be achieved by:

```
args=8
x=24
f=28
pt=36
```

However, the `.dsect` block is more readable and more clearly defined. This mechanism also lends itself to defining offsets which match C structure definitions. The `.dsect` block above could declare offsets for the following C structure:

```
struct {
    int unused[2];
    int args[4];
    int x;
    double f;
    struct { int x, y, z, w; } pt;
} ss;
```

Suppose the address of `ss` were loaded into register `r3`. The fields might be accessed by:

```
lwz r0, x(r3)      # load ss.x
stfd fr0, f(r3)    # store ss.f
lwz r5, pt+4(r3)   # load 2nd word of ss.pt
```

Global Variables

Global, or file scope, variables are accessed in one of two ways, depending on whether they are in SDA (see “Special Data Area Optimizations” on page 137). Symbols that are not in SDA cannot be accessed directly, due to the 16-bit load/store offset range of the PowerPC architecture.

Instead, accessing a non-SDA variable is a two-step process. Two instructions are used, generally with the instruction `lis` to get the high 16 bits of the variable’s address into a register, followed by a `load` or `store` instruction using that register as an index register, offset by the low 16 bits of the variable’s address:

```
.text
.extern x
lis r3, %hiadj(x)      # high part of address of x
lwz r4, %lo(x)(r3)     # value of x
addi r4, r4, 1         # increment x
stw r4, %lo(x)(r3)     # store updated value of x
```

Note that `%hiadj` must be used instead of `%hi` to load the high 16 bits in this case. This is because the register+offset addressing mode sign extends the offset, and `%hiadj` compensates for this.

Variables in the ROM or RAM SDAs allow a different method for accessing their values. The technique above will still work, but the SDA gives the advantage of allowing shorter accesses. An example of accessing a variable `y` in the `.sdata` section is:

```
.text
lwz r4, %sdaoff(y)(r13) # load y's value to r4
addi r4, r4, 1          # increment
stw r4, %sdaoff(y)(r13) # store updated value of y
                        #      [...]
.section ".sdata", "aw"

y:
    .long 0
```

where `y` uses the `%sdaoff()` operator. This operator takes a relocatable symbol and causes the assembler to output a special relocation type that the linker will resolve as the offset from the base address for the Small Data Area. Register `r13` is the base register for the RAM SDA, so this process will produce the correct address for `y`.

The `%sdaoff()` operator also references variables that are put into the ROM SDA area, as in the example above. However, the register `r2` should be used as the base register instead of `r13`, and the variables should be declared in the `.sdata2` section rather than the `.sdata` section.

Zero-Initialized Variables

Zero-initialized variables can be declared with the `.comm` or `.sbss` directive, depending on whether they should go into the RAM SDA:

```
.comm    z, 24
.sbss    sz, 4
```

This code will declare `z` as a 24-byte zero-initialized variable in the `.bss` section, and `sz` as a 4-byte zero-initialized variable in the `.sbss` section within the RAM SDA.

Writing Portable Code

Many compiler vendors have implemented the C and C++ languages for a wide variety of system architectures. One important reason for this is that these languages greatly simplify the task of building and maintaining software for multiple platforms. However, not all features of C and C++ accommodate this goal. In fact, both languages intentionally provide features that perform differently on different systems. For a program to be truly portable, the programmer must avoid these non-portable features of the C and C++ languages. This chapter discusses some of the non-portable assumptions that can cause programs to fail and how to avoid portability problems with Green Hills C/C++ compilers.

The C/C++ language specifications define programs in such a way that portable programs will always work with all appropriate language compilers, including any Green Hills C/C++ compiler. However, difficulties arise when programmers make non-portable assumptions about the machine or compiler that they are using. As a result, a program may appear to compile and operate correctly when compiled with one vendor's compiler, but not with a Green Hills compiler.

Certain differences between compilers are specific to the individual compiler vendor. In addition, many more differences are particular to the processor and the operating system being used. To avoid these differences when porting

between different system architectures, portable code should be consistently written.

Compatibility Between Green Hills Compilers

All Green Hills C/C++ compilers follow the same interpretation of the C/C++ languages as described in this manual; however, some Green Hills C or C++ compilers do not include certain features or options.

For information about compatibility with Green Hills compilers for other languages, see “Mixing Languages” on page 764.

Word Size Differences

Green Hills compilers support machines with 32-bit and 64-bit word size. Porting C/C++ programs between machines of different word sizes requires particular care because word size may affect most primary data types.

Some machines are *byte addressable*, meaning that their addresses refer to 8-bit bytes. Typically, byte addressable machines operate on 8, 16, 32, 64, and 128-bit quantities. Other machines are *word addressable*, meaning that their addresses refer to words of a standard size varying from 16 to 64 bits. Word addressable machines typically operate on multiples of the word size.

A program that operates on a machine of one word size may not operate on a machine of a different word size. Thus, word size incompatibility problems may arise if two different machines have different word sizes, or if one machine is word addressable and the other is byte addressable. The word size affects the range of numbers implemented by integer data types, as well as the precision and range of single and double precision floating-point data types.

The most common word size problems are integer and floating-point underflows, overflows, and loss of precision. The layout of bit aligned data structures will vary with the word size, so overlaying structures in memory makes programs difficult to port to another compiler. Address arithmetic done in integer variables is often not portable.

Range of Representable Values

The size of each basic numeric type controls the range of values which may be represented by that type. The header files `<limits.h>` and `<float.h>` provide defined symbols which represent the minimum and maximum values for all numeric data types in C/C++. A portable program should use these symbols and never depend on the use of values outside the allowed range.

If arithmetic operations cause overflow, underflow, or loss of precision, the program may not detect the error or may behave differently on different systems.

Relative Sizes of Data Types

Both C and C++ place very weak requirements on the relative size of the basic types, though programs in either language commonly assume otherwise. For example, both languages only require that `short` be no larger than `int` and that `int` be no larger than `long`. It would be legal for `short`, `int`, and `long` to all be the same size or for them all to be different. For instance, with some 32-bit Green Hills C/C++ compilers, `short` is 16 bits and both `int` and `long` are 32 bits. The assumption that `int` is twice as large as `short`, but the same size as `long` is non-portable, because, with 64-bit Green Hills C/C++ compilers, `short` is 16 bits, and `int` and `long` are either 32 or 64 bits.

Another common non-portable assumption is that pointers are the same size as `int` or `long`. Neither is guaranteed. With all 32-bit Green Hills C/C++ compilers, pointers are 32 bits. But with 64-bit Green Hills C/C++ compilers, pointers may be either 32 or 64 bits, independent of the size of `int` or `long`. Use the `uintptr_t` and `intptr_t` types provided by `stdint.h` as portable integer types large enough to represent any data pointer, but not function pointers.

In both languages, all integer constants have type `int` unless marked with a type suffix. In certain cases, the use of a plain integer constant instead of a long integer constant may be non-portable.

Endianness Problems

Programs that overlay characters and integers in memory or that use character pointers to integer variables and vice versa are often not portable between machines with different byte ordering.

Programs that declare a single variable with different integer types in different modules may fail when ported to a machine with a different byte order.

Alignment Requirements

Some systems will not load or store a two byte object unless that object is on an even address. Other systems have a similar requirement for four or eight byte objects. Others may allow certain accesses, but require more time to perform them. Therefore, alignment of data is both a matter of correctness and of time efficiency. Although increased alignment may improve performance, it also consumes space, due to padding inserted to achieve alignment.

The alignment requirements on each system are chosen both to satisfy the restrictions of the hardware and to achieve a reasonable balance between performance and space. The alignment rules are often not configurable and they differ for each system. Thus, programs that make assumptions about the relative position of data objects in memory or elements within classes, structures, or arrays are not portable—even among the Green Hills C++ compilers.

The C/C++ languages impose the following restrictions on size and alignment:

- The alignment of a structure, union, class, or array is equal to or greater than the maximum alignment requirement of any of its members.
- The size of a structure, union, class, or array is always a multiple of the maximum alignment requirement of any of its members.
- The offset of any member of a structure, union, class, or array is always a multiple of its alignment requirement.
- All dynamic memory allocation routines provided with the compiler will return a pointer aligned to the maximum alignment for any object on that machine.

All Green Hills C/C++ compilers also satisfy these principles:

- The stack is maintained on an alignment suitable for any object.
- Parameters and local variables are allocated on the stack according to their alignment requirement.
- Local variables are arranged on the stack to avoid unnecessary padding due to alignment.

Structures, Unions, Classes, and Bitfields

The preceding issues of size, byte order, and alignment all affect the allocation of data in memory. In particular, compound data structures such as unions, structures, classes, bitfields, and arrays are very much affected by these issues.

Unions

A union allows the same memory location to be accessed as more than one type. This is inherently non-portable. Suppose a union consists of an integer and an array of four characters. Whether the first element of the array is the most or least significant part of the integer depends on byte order. It is not even certain that the integer and the array of characters have the same size.

These problems increase when a program combines integer, floating-point, and pointer fields, and are even more severe when unions consist of structures or bitfields.

Structures and Classes

Green Hills C/C++ compilers always allocate fields in a structure or class in the order in which the program declares them.

The exact offset of each field from the base of the structure/class depends on the size and alignment of the field itself and of those which precede it. The offset of the first field is always 0 in C, but not necessarily in C++ (for instance with multiple inheritance). Also, padding is inserted as necessary to satisfy the alignment requirement of each subsequent field, and may also be added at the end of the structure/class to make its overall size a multiple of its alignment.

Any program is non-portable if it assumes the offset of a field within a class/structure or if it assumes that certain fields in two different classes/structures always have the same offset.

Bitfields

The allocation of bitfields in a structure is dependent upon alignment rules. Additionally, the exact layout of bits within a bitfield varies between systems and cannot be assumed by a portable program.

Assumptions About Function Calling Conventions

Early implementations of C used a very straightforward approach to function calls. All parameters were pushed on the stack from right to left. All integral types smaller than `int` were promoted to `int` and `float` was promoted to `double`. Given this implementation, it was possible to write functions in C which handled variable parameters, even before the `<stdarg.h>` facilities. But such functions are non-portable and depend on an intimate knowledge of the calling conventions.

Many modern C/C++ compilers pass some parameters in registers and may not evaluate parameters from right to left. Integer and floating-point variables, not to mention structures, may have different rules. One non-portable assumption is that a `double` may be passed to a function which expects two integers. Not only does this assume a relationship between the size of the two types and a certain ordering of bytes and words, but it assumes that doubles and integers follow all of the same rules. Both of these assumptions are erroneous.

A much more common assumption is that pointer and integers may be interchanged when passing parameters. Neither C nor C++ guarantee that a pointer will be assigned to an integer and back without loss of information—even if the pointer and the integer are the same size. The only safe way to write a function that can correctly accept either pointer or integer parameters is to use the `<stdarg.h>` facilities.

Even among integral types, a program may assume that `int` and `long` are interchangeable. A C program can invoke `printf` using the `%d` operator to refer to a `long` parameter; however, this program will fail when ported to a system where `long` is larger than `int`. The correct way is to use `%ld` for `long` parameters.

The same portability problems exist with respect to function return values. A function which returns an `int` should never be used to return a pointer or `long` or floating-point value, even though it may work reliably on a particular system. A common mistake is to omit a declaration of a function that returns a pointer, and then place a cast around the invocation of that function. The cast cannot fix the error; it only prevents the compiler from reporting it.

Pointer Issues

Nearly all machines supported by Green Hills compilers are byte addressable, but neither C nor C++ require this. On some machines, a pointer to an `int` and a pointer to a `char` are not interchangeable. ANSI C requires that void pointers handle all pointer types, but the void pointer must be cast or assigned to its original type before being used. Similarly, C does not require that function pointers and data pointers be interchangeable, but some C programs incorrectly make this assumption.

C supports portable pointer arithmetic, provided it is used correctly. In ANSI C, the difference of two pointers is only defined for cases where both pointers refer to two elements within the same array object. Even so, in some unusual cases the difference may be outside the range of `ptrdiff_t` (which is either `int` or `long`). Subtraction or comparison of pointers to two separate objects may give non-portable results due to differences in memory layout or because pointers are signed on one system and unsigned on another.

Pointer arithmetic should always be done directly on the pointers when possible. If you need an integer type, use `uintptr_t` or `intptr_t`.

NULL Pointer

In all Green Hills C/C++ compilers, and most C compilers in general, the NULL pointer has the value 0. But there are still two portability issues. One issue is that some older programs depend upon the contents of memory location 0 being 0. This is now a well recognized programming error and some modern machines purposely give a memory fault for any attempt to read or write to location 0.

A more subtle problem is the size of NULL. On a machine where pointers are larger than `int`, it is incorrect to use the constant 0 as a NULL pointer, because 0 is of type `int`, which is smaller than a pointer. This matters when passing NULL to a function that takes variable parameters or is not declared with a prototype.

Character Set Dependencies

Not all computer systems use the same characters. Although all computer systems recognize letters, digits, and the standard punctuation characters, considerable variation exists among the less commonly used characters. Therefore, programs which use the less common characters may not be portable.

Green Hills compilers use the ASCII character set and the ASCII collating sequence. Some language implementations use a different *collating sequence*, such as EBCDIC. Programs which manipulate character data, especially string sorting algorithms, may be dependent on a particular character collating sequence (the order in which characters are defined by the implementation). If one character appears before a second character in the collating sequence, then the first character will be considered smaller than the second character when they are compared. In the ASCII collating sequence, the lowercase letters 'a' to 'z' appear as the contiguous integer values 97 to 122 (decimal). In other collating sequences, such as EBCDIC, the lowercase letters are not contiguous.

To make character and string sorting programs portable, dependencies on the character collating sequence should be avoided. If a program is designed to operate with a collating sequence other than ASCII, it may be necessary to modify string and character comparison code in order to operate with ASCII.

Floating Point Range and Accuracy

The range, precision, accuracy, and base of floating-point arithmetic can vary widely between machines. This can lead to many portability problems that can only be addressed numerically. Green Hills compilers use the standard IEEE floating-point representation.

Operating System Dependencies

The file and I/O device naming conventions vary greatly among computer systems. Thus, programs that access operating system resources (such as files) by their system names are often not portable. In order to write portable programs, the use of explicit filenames in the program should be minimized. Preferably, these names can be input to the program when the program is run.

If a program contains explicit filenames it may be necessary to change them to names acceptable to the target system. Refer to your target operating system documentation for a description of legal filenames for that environment.

Assembly Language Interfaces

Programs which use embedded assembly code or interface to external assembly will require all of the assembly code to be rewritten when the program is transported to a new machine.

Evaluation Order

None of the language specifications fully describe the order in which the various components of an expression or statement must be evaluated. Additionally, the language specifications disallow computations whose results depend upon which permitted evaluation order is used. Many illegal programs have gone undetected because they have only been compiled with one compiler. Because the evaluation order may differ between compilers, some illegal programs may not operate as expected when compiled with a Green Hills compiler.

Some language implementations may evaluate the arguments to a function from right to left, others from left to right.

Green Hills compilers may execute expressions with side effects (such as subroutine, procedure, or function calls) in a different order than other compilers. When a variable is modified as a side effect of an expression, and its value is also used at another point in the expression, it is not specified whether the value used at either point in the expression is the value before or after modification. Different values for the same variable could potentially be used at different places in the expression, depending on the evaluation order observed by the compiler.

Green Hills compilers may also execute operators with side effects (such as ++, --, or +=) in a different order from that of other compilers.

Machine-Specific Arithmetic

Certain arithmetic operators in C/C++ are intended to generate the most efficient corresponding operation on the target machine. If all input values are within the expected range, the results are portable. Out of range values may give different results on different systems.

Shift

The shift operators in C/C++ possess machine-specific arithmetic characteristics. If the right-hand operand is negative or is greater than or equal to the number of bits in the promoted left-hand operand, the behavior is undefined and the results vary among compilers and hardware, and depending on whether the right hand operand is constant.

If the left-hand operand of a right shift is signed, C/C++ does not require the compiler to propagate the sign bit. While all Green Hills compilers propagate the sign bit, a conforming compiler is allowed to propagate zeros into the high bits.

Division

Division always satisfies the following rule:

$$(a / b) * b + a \% b == a$$

When either *a* or *b* is negative, a C89 or C++ compiler is allowed to round *a / b* either toward zero or toward negative infinity. In C99, a compiler must always round toward zero. Green Hills always rounds toward zero, but a non-GHS compiler may round toward negative infinity.

Illegal Assumptions About Compiler Optimizations

Some programs illegally depend on the exact code generated by a particular compiler. Such programs are particularly difficult to port to an advanced optimizing compiler because the optimizer makes major changes in the code in order to reduce the program's size and increase the program's speed. This section describes some of the most common illegal assumptions made about code generation. The optimizations referenced in this chapter are described in greater detail in Chapter 7, "Optimizing Your Programs".

Implied Register Usage

Some programs rely on the exact register allocation scheme used by the compiler. Such programs are illegal and will require modification before they can be transported.

For example, C/C++ programs that rely on register variables being allocated sequentially to pass hidden parameters will not work. Hidden returns (i.e., using `return` and expecting to return the value of the last evaluated expression) will not work either.

Memory Allocation Assumptions

Green Hills compilers allocate memory in a different manner than that of typical compilers. To ensure portability, programs compiled with a Green Hills compiler must not make assumptions regarding the order or allocation of variables in memory except where the language standard specifies it. If a program depends on the illegal memory allocation peculiarities of a particular compiler, it may experience portability problems. Note the following:

- Green Hills compilers do not necessarily allocate variables in the order of declaration.
- Green Hills compilers may not allocate unused variables.
- Green Hills compilers allocate certain variables to registers that standard compilers would always allocate to memory.

Memory Optimization Restrictions

This section is very important when the system code or application uses shared memory or signals.

Using the **Advanced→Optimization Options→Memory Optimization** option (-OM) enables the compiler to assume that memory locations do not change asynchronously with respect to the running program (see “Memory Optimization” on page 359). In particular, when the compiler reads or writes to some memory location, it assumes that the same value remains there several instructions later. To avoid the (potentially high) speed penalties involved in re-reading memory, the compiler searches a register for a copy of the value to use instead.

If the `volatile` keyword is not used correctly, this option may cause problems for many parts of operating systems, device drivers, memory mapped I/O locations, shared memory environments, multiple process environments, and interrupt-driven routines, and when Linux/Solaris style signals are enabled.

An example of the potential problems involved with memory optimizations is that many Linux/Solaris device drivers need to use memory locations which are really I/O registers that can change at any time. A typical example of a loop waiting for a device register to change is:

```
while (!( *TSRADDR & (1 << TXSBIT) ));
```

If memory optimizations are enabled while compiling this loop, the compiler may generate code that reads the value pointed to by `TSRADDR` only once. If the **Advanced→Optimization Options→Optimize All Appropriate Loops** option (**-OL**) is also enabled, it is almost certain that this will be the case. When this happens, the loop executes either once or forever (depending on the value of the bit when it is first tested) and will be rendered either ineffective or fatal. Depending on the situation, the compiler may detect such loops and generate code that operates correctly even with **Advanced→Optimization Options→Memory Optimization** option (**-OM**) enabled. However, if the loop body were to test more than one bit at the same address, the compiler may distort the loop in an attempt to read memory as few times as possible.

The compiler assumes that the `volatile` type qualifier is used when it is available. This means that any **Optimization→Optimization Strategy** implies **Advanced→Optimization Options→Memory Optimization** in C++ or in the ANSI modes of C. If circumstances prevent the use of `volatile`, you should disable the **Advanced→Optimization Options→Memory Optimization** option (**-Onomemory**). Note that this will leave the other general optimizations enabled (see “Optimization Options” on page 176).

Problems with Source Level Debuggers

Variable Allocation

Once a variable is allocated to a register it will always reside in that register. However, because other variables may share the register, it may not always contain the current value of the variable. This may cause a source level debugger to give incorrect results. If a variable is looked at outside the range of its use, the compiler may have temporarily allocated that register for some other purpose. The best strategy is to check results just after they are assigned, or when the current value is going to be used later. Near the end of a function, most of the local variables will no longer be in use, so it is more likely that the register has been re-allocated.

Problems with Compiler Memory Size

The compiler primarily uses memory for the program, static data structures, global declarations, parse trees, and generated machine code. Global declarations consist of the global constant, type, variable, and function

declarations. Memory usage increases when a compilation includes large numbers of declarations. Even unused global declarations must be stored throughout the compilation. If memory size problems exist, reduce the size of the **include** files by including only the necessary declarations.

Basic blocks also require memory. Every possible branch creates a new block. Memory usage may increase on machine-generated programs with very large `switch` statements or with a very large number of small `if` statements.

The C and C++ compilers store the entire program into a parse tree before proceeding with optimization and code generation. The optimizer modifies the parse tree and then passes it on to the code generator. The code generator produces an internal representation of the machine code to be output for the function. Another optimization phase is then called to modify this machine code. Finally, the optimized machine code for the function is output. After the machine code is output, the memory being used for the parse tree, optimization, and machine code generation is released for use in processing the next function.

Usually, the size and complexity of the largest function in the program determine the maximum memory usage for parse trees and machine code. If memory size problems exist, turn off the optimizer and reduce the size or complexity of the relevant functions. A simple function of less than 100 lines should not cause memory size problems. However, procedures containing very complex statements, or more than 1000 lines of code, may require several megabytes of memory in order to compile.

Enhanced asm Macro Facility for C and C++

Chapter 19

This Chapter Contains:

- Definition of Terms
- Using and Defining the asm Macro Facility
- Using asm Macros
- PowerPC asm Macros
- Guidelines for Writing asm Macros

Although having the ability to write portable code is an advantage of using the C language, sometimes it is necessary to introduce machine-specific assembly language instructions into C code. This need arises most often within operating system code that must deal with hardware registers that would otherwise be inaccessible from C. You can use the *asm* facility to include this assembly code.

In earlier versions of C, the *asm* facility included a line that looked like a call on the function **asm** that took one argument, a string:

```
asm ( "assembly instruction here" ) ;
```

Unfortunately, this technique has shortcomings when the assembly instruction needs to reference C language operands. The user has to guess the register or stack location into which the compiler would put the operand and encode that location into the instruction. If the compiler's allocation scheme changes or, more likely, if the C code surrounding *asm* () changes, the correct location for the operand in *asm* () also changes. The user has to be aware that the C code would affect *asm* () and change it accordingly.

In addition, *asm* () statements that contain labels or other unique identifiers might cause errors if you enable certain optimizations. You may need to use temporary labels to prevent these errors (See "Labels" on page 407 for more information).

The new facility presented here is compatible with old code, since it retains the old capability. In addition, you can define *asm* macros that describe how machine instructions should be generated when their operands take particular forms that the compiler recognizes, such as register or stack variables.

Although this enhanced *asm* facility is easier to use than before, the user is still strongly discouraged from using it for routine applications, because those applications will not be portable to different machines. The *asm* facility helps implement operating systems in a clean way.

Optimizations may work incorrectly on C programs that use the *asm* macro facility, leading to compile-time or run-time errors. This is more likely when the *asm* macros contain instructions or labels that are unlike those that the C compiler generates. No compiler warning or error messages will be issued. Furthermore, the user may need to rewrite *asm* code in the future, in order to maximize its benefits as new optimization technology is introduced into the compilation system.

Wherever possible, consider using intrinsic functions instead of `asm` macros. `asm` macros act as an optimization barrier to the compiler, since the compiler is unable to interpret the actions or understand the instructions in the `asm` macro. Intrinsic functions, on the other hand, are understood by the compiler, so the compiler need not make worst-case assumptions about the behavior of the instructions. Intrinsic functions are also easier to use since the compiler takes care of loading and storing the operands automatically, as necessary.

Definition of Terms

The following terms are used when describing the `asm` macro facility.

Term	Definition
<code>asm</code> macro	The mechanism by which programs use the enhanced <code>asm</code> facility. An <code>asm</code> macro has a definition and uses. The definition includes a set of pattern/body pairs. Each pattern describes the storage modes that the actual arguments must match for the <code>asm</code> macro body to be expanded. The uses resemble C function calls.
storage mode	The compiler's idea of where the argument can be found at run time. Examples are "in a register" or "in memory."
pattern	Specifies the modes for each of the arguments of an <code>asm</code> macro. When the modes in the pattern all match those of the use, the corresponding body is expanded.
<code>asm</code> macro body	The portion of code that will be expanded by the compiler when the corresponding pattern matches. The body may contain references to the formal parameters, in which case the compiler substitutes the assembly language location for the corresponding parameter.

Using and Defining the `asm` Macro Facility

The following is a general example intended to demonstrate how to define and use an `asm` macro. For an example specific to PowerPC, see "PowerPC `asm` Macros" on page 795.

Suppose your machine has an instruction that takes one operand, which must be in a register. It may be convenient to have a function that produces inline code to use this instruction, and works with register variables or constants.

This example consists of two parts: the definition of the `asm` macro and its use.

Pseudo-Code Definition

Define an `asm` macro, called `ASM`:

```
asm void ASM(newpri)
{
%reg newpri
    asm_instruction newpri
%con newpri
    asm_move newpri, temp_register
    asm_instruction temp_register
}
```

The lines that begin with `%` are patterns. If the arguments at the time the macro is called match the storage modes in a pattern, the code that follows the pattern line will expand.

Use

The table below shows the (assembly) code that the compiler will generate with two different uses of the assembly instruction. It uses the following introductory code (along with the above definition):

```
void f()
{
    register int i = 3;
    ASM(i);
    ASM(3);
}
```

Code	Matches	Generates
<code>ASM(i);</code>	<code>% reg</code>	<code>asm_instruction register_i_is_in</code>
<code>ASM(3);</code>	<code>% con</code>	<code>asm_move newpri -> temp_register</code> <code>asm_instruction temp_register</code>

The first use of `ASM` has a register variable as its argument (assuming that `i` actually gets allocated to a register). This argument has a storage mode that matches `reg`, the storage mode in the first pattern. Therefore, the compiler expands the first code body. `newpri`, the formal parameter in the definition, has been replaced in the expanded code by the compiler's idea of

the assembly-time name for the variable `i` (*register_i_is_in*). Similarly, the second use of `ASM` has a constant as its argument, which leads to the compiler's choosing the second pattern. Here again `newpri` has been replaced by the assembly-time form for the constant 3.

Using asm Macros

You can use the enhanced `asm` facility to define `asm` macros. `asm` macros are constructs that behave syntactically like static C functions. Each `asm` macro has one definition and zero or more uses per source file. You must put the macro's definition in the file that uses the macro (or include the definition with `#include`). You can define the same `asm` macro multiple times (and differently) in several files.

The `asm` macro definition declares a return type for the macro code, specifies patterns for the formal parameters, and provides bodies of code to expand when the patterns match. When the compiler encounters an `asm` macro call, it replaces each use of a formal parameter with an assembly language location as it expands the code body. This constitutes an important difference between C functions and `asm` macros. In some cases, an `asm` macro can change the values of its arguments, whereas a C function can only change a copy of its arguments.

If you pass any of the following as an argument to an `asm` macro, the macro can read and modify that argument directly:

- Structures contained in registers.
- Scalar variables that the compiler would not promote if passed to a function without a prototype.

For all other types of arguments, the compiler creates a temporary variable and passes that variable to the `asm` macro instead. Whenever the compiler uses a temporary variable, modifying the value directly within the `asm` macro body does not affect the value of the original argument. These temporary variables are eligible for register allocation and are allocated to registers if the register allocator has sufficient physical registers available to it:

- Function names
- Structures contained in memory
- Structure and array elements

- Compound expressions
- Addresses
- Scalar variables that the compiler would promote if passed to a function without a prototype



Note The compiler promotes the type of the scalar variable as if you were passing it to a function without a prototype in C. The compiler does this whether or not you specify the asm macro's parameter list in a prototype form. This means that the compiler promotes variables of type unsigned char, signed char, signed short, unsigned short, and float, and then passes them to the macro through temporary variables.

Calls to asm macros look exactly like normal C function calls. You can use them in expressions and they can return values. The arguments to an asm macro can be arbitrary expressions, except that they must not contain uses of the same or other asm macros. Return values are passed according to the target-specific calling convention.

The following pseudo-code shows how passing addresses works:

```
asm void make10(addr)
{
    %reg addr
        asm_move 10 -> temp_register
        asm_store temp_register -> *addr
    %nearmem addr
        asm_load addr -> temp_register_1
        asm_move 10 -> temp_register_2
        asm_store temp_register_2 -> *temp_register_1
    %error
}
void func()
{
    int i, array[10];
    make10(&i);
    make10(array);
    make10(&array[3]);
}
```

Definition

The syntactic classes *type_specifier*, *identifier*, and *parameter_list* are presented in the style used in C-language compilers. A syntactic description enclosed in

square brackets ([]) is optional, unless the right bracket is followed by *, which means “zero or more repetitions.” For example:

```
asm [type_specifier] identifier ( [parameter_list] )
{
    [storage_mode_specification_line
        asm_body] *
}
```

An asm macro consists of the keyword `asm` followed by what looks like a C function declaration. Inside the macro body there are zero or more pairs of *storage_mode_specification_lines* (patterns) and corresponding *asm_bodies*. If the *type_specifier* is other than `void`, the asm macro should return a value of the declared type.

Instead of the `asm` keyword, one may also use the `__asmleaf` keyword. This affects what registers are available to the programmer to use for scratch purposes. See “Guidelines for Writing asm Macros” on page 798 for more information.

storage_mode_specification_line

```
% [storage_mode [identifier [, identifier] * ] ] *
```

A *storage_mode_specification_line* consists of a single line (no continuation with the backslash character (\) is permitted) that begins with a percent sign (%) and contains the names (*identifiers*) and *storage modes* of the formal parameters. Modes for all formal parameters must be given in each *storage_mode_specification_line* (except for error). The percent sign (%) must be the first character on a line (including whitespace). If an asm macro has no *parameter_list*, the *storage_mode_specification_line* may be omitted.

Storage Modes

These are the storage modes that the compiler recognizes in asm macros:

Storage mode	Description
<code>con</code>	A compile-time constant.
<code>error</code>	Generates a compiler error. You can use this mode to flag errors at compile time if no appropriate pattern exists for a set of actual arguments.

Storage mode	Description
<code>farsprel</code>	A location on the stack that is too far away to be accessed in a single instruction.
<code>lab</code>	A compiler-generated unique label. The <i>identifier(s)</i> that are specified as being of mode <code>lab</code> do not appear as formal parameters in the <code>asm</code> macro definition, unlike the preceding modes. Such identifiers will be expanded to be unique. Example: <pre>asm void check_for_bit_set(r) { %reg r %lab endlab b_if_not_bit_set r, endlab software_trap endlab: %error } void check_status(int i, int j) { /* Using the macro twice would cause the "endlab" label to be * defined twice if it was not specified as unique with %lab */ check_for_bit_set(i); check_for_bit_set(j); }</pre>
<code>mem</code>	Matches any allowed machine addressing mode, with the exception of <code>reg</code> and <code>con</code> .
<code>farmem</code>	A location in memory that cannot be accessed with a single load instruction. This may need to be broken up into multiple instructions, and the steps for doing this may depend on the compilation mode.
<code>nearmem</code>	A location in memory that can be accessed with a single load instruction (such as nearby stack variables or variables in a Small Data Area).
<code>reg</code>	A <code>treg</code> or <code>ureg</code> .
<code>treg</code>	A compiler-selected temporary register.
<code>ureg</code>	A C register variable that the compiler has allocated in a machine register.

asm Body

The `asm` body represents the (presumed) assembly code that the compiler will generate when the modes of all of the formal parameters match the associated pattern. Syntactically, the `asm` body consists of the text between two pattern

lines that begin with a percent sign (%) or between the last pattern line and the right curly brace (}) that ends the asm macro. C-language comment lines are not recognized as such in the asm body. Instead, they are simply considered part of the text to be expanded.

Formal parameter names may appear in any context in the asm body, delimited by non-alphanumeric characters. For each instance of a formal parameter in the asm body the compiler substitutes the appropriate assembly language operand syntax that will access the actual argument at run time. As an example, if one of the actual arguments to an asm macro is `x`, an automatic variable, a string like `4(%fp)` would be substituted for occurrences of the corresponding formal parameter. An important consequence of this macro substitution behavior is that asm macros can change the value of their arguments. This differs from standard C semantics.

For `lab` parameters, a unique label is chosen for each new expansion.

If an asm macro is declared to return a value, it must be coded to return a value of the proper type in the machine register that is appropriate for the implementation.

An implementation restriction requires that no line in the asm body may start with a percent sign (%).

The PowerPC compiler also supports the following primitives in the body of an asm statement:

`%SPOFF( m )`

If `m` is of the `farsprel` storage class, this expands to an integer containing its offset from the stack pointer. For example:

```
%farsprel m
addis    r3, sp, %hiadj(%SPOFF(m))
lwz      r3, %lo(%SPOFF(m)) (r3)
```

PowerPC asm Macros

The following is an example of how to use asm macros for the PowerPC architecture. This example shows an operand multiplied by the value 10.

The first item to notice is that separate cases for PID and non-PID modes are needed, but that a single mode, `%nearmem`, handles SDA, ZDA, and near stack references. It is assumed that constants can be loaded in a single `li` instruction; if this is not the case, the `%hiadj()/%lo()` pairs must be used for the entire construct. Alternatively, users can refrain from passing constants, and the `%error` directive can be used to check for non-conformance. Notice that `%farsprel` must be handled before `%farmem`, since `%farmem` is a superset of `%farsprel`.

On the PowerPC, any of the argument registers may be used as temporary (scratch) registers in an `asm` macro. `r3` is the return register, so the return value is placed there.

C preprocessor directives may not be used within an `asm` macro. Thus, much of the code for the PID and non-PID cases is copied over rather than only changing the relevant parts.

Also note that the number of possible cases for an `asm` macro goes up exponentially with the number of general arguments. `asm` macros are designed to be used in specific cases where the desired usage patterns are well understood.

```
#ifdef __ghs_pid
asm int mult10(a) {
%reg a
    mulli    r3, a, 10
%con a
    li       r3, a
    mulli    r3, r3, 10
%farsprel a
    addis    r3, sp, %hiadj(%SPOFF(a))
    lwz      r3, %lo(%SPOFF(a))(r3)
    mulli    r3, r3, 10
%nearmem a
    lwz      r3, a
    mulli    r3, r3, 10
%farmem a
    addis    r3, r13, %pidhiadj(a)
    lwz      r3, %pidlo(a)(r3)
    mulli    r3, r3, 10
%error
}
#else /* not in PID mode */
asm int mult10(a)
{
%reg a
    mulli    r3, a, 10
%con a
    li r3, a
    mulli r3, r3, 10
%farsprel a
    addis    r3, sp, %hiadj(%SPOFF(a))
    lwz      r3, %lo(%SPOFF(a))(r3)
    mulli    r3, r3, 10
%nearmem a
    lwz      r3, a
    mulli    r3, r3, 10
%farmem a
    lis      r3, %hiadj(a)
    lwz      r3, %lo(a)(r3)
    mulli    r3, r3, 10
%error
}
#endif
int memory_variable = 3;
void foo(int *proc) {
}
int main(void){
    int stack_variable = 4;
    int local_variable = 5;
    foo(&stack_variable); /* force stack_variable on the stack */
    if ((mult10(memory_variable) != 30) ||
        (mult10(stack_variable) != 40) ||
        (mult10(local_variable) != 50))
        printf("Asmmacro error in %s\n", __FILE__);
    return 0;
}
```

Guidelines for Writing asm Macros

Here are some guidelines for writing asm macros.

- Use intrinsic functions preferentially over asm macros. Intrinsic functions do not inhibit optimizations the same way that asm macros may, and they are easier to use.
- Know the implementation. You must be familiar with the C compiler and assembly language with which you are working. You can consult the Application Binary Interface for your machine for the details of function calling and register usage conventions.
- Observe register conventions. An asm macro can alter temporary registers at will, but the permanent registers should not be written to directly, as the arguments to asm macros might be allocated to permanent registers. However, you may write to any formal parameter, regardless of whether the compiler automatically allocated it to a register. You must know in which register or registers the compiler returns function results.

An asm macro declared with the `__asmleaf` keyword can only alter the return registers. This allows the compiler to generate better code since the other temporary registers will not be assumed to be modified across the asm macro.

- Handle return values. asm macros may “return” values. That means they behave as if they were actually functions that had been called via the usual function call mechanism. asm macros must therefore mimic C’s behavior in that respect, returning values in the same way as normal C functions. Float and double results sometimes get returned in different registers from integer results. On some machine architectures, C functions return pointers in different registers from those used for scalars. Finally, structures may be returned in a variety of implementation-dependent ways.
- Cover all cases. The asm macro patterns should cover all combinations of storage modes of the parameters. The minimal set that covers all cases is `reg`, `con`, and `mem`. The compiler attempts to match patterns in the order of their appearance in the asm macro definition. If the compiler encounters a storage mode of `error` while attempting to find a matching pattern, it generates a compile-time error for that particular asm macro call.
- Beware of argument handling. asm macro arguments are used for macro substitution. Thus, unlike normal C functions, asm macros can alter the underlying values to which their arguments refer. Altering argument values

makes it impossible to substitute an equivalent C function call for the `asm` macro call.

- `asm` macros are inherently non-portable and implementation-dependent. Although they make it easier to introduce assembly code reliably into C code, the process cannot be made foolproof. You will always need to verify correct behavior by inspection and testing.
- Debuggers will generally have difficulty with `asm` macros. You cannot set source-level breakpoints in the `asm` body.
- Make sure that storage mode specification lines begin with `%` in the first column. There must not be any whitespace before this character.

The ELF File Format

Chapter 20

This Chapter Contains:

- Formats of Relocatable and Executable Files
- 32-Bit ELF Data Types
- ELF Headers
- ELF Identification
- ELF Sections
- Symbol Tables
- String Tables
- Program Headers

This chapter explains the formats of ELF (Executable and Linking Format) files. Sections of this chapter have been reproduced with permission from UNIX System Laboratories, Inc. For more information about ELF files, see *System V Application Binary Interface*, 1993, UNIX System Laboratories, Inc., published by Prentice-Hall, Inc.

This chapter deals only with the 32-bit ELF format. Programs and object files built for 64-bit targets use the 64-bit ELF format, which is similar to the 32-bit format, but uses larger fields to store some values.

Formats of Relocatable and Executable Files

ELF object files are the standard output of the compiler, assembler, and linker. An ELF file can be either of the following:

- a *relocatable object file* — which contains program code and data and is suitable for linking with other object files.
- an *executable file* — which contains a program suitable for execution.

The following diagram shows the differences in format between these kinds of ELF files:

Relocatable file	Executable file
ELF Header	ELF Header
Program Header Table (<i>optional</i>)	Program Header Table
Section 1	Segment 1
. . .	
Section <i>n</i>	Segment 2
. . .	
. . .	. . .
Section Header Table	Section Header Table (<i>optional</i>)

Both forms of ELF file begin with an *ELF Header*, which serves as the table of contents. All other data and tables in the file may appear in any order.

An executable file must have a *Program Header Table*, which tells the system how to load the program. This component is optional in a relocatable file.

A relocatable file requires a *Section Header Table*, containing information about its *sections* in order to be subsequently linked. Each section has an entry in the table, giving information such as the section's name and size. Sections form the majority of the relocatable object file, and comprise such information as instructions, data, symbol table, and relocation information.

32-Bit ELF Data Types

The ELF file format supports 32-bit architectures with 8-bit bytes. It must be modified for 64-bit architectures. ELF files represent some control data with a machine-independent format to identify object files and interpret their contents. Part of an ELF file uses the encoding of the target processor, regardless of the machine on which the file was created:

Type name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

All data structures that the ELF file format defines follow the usual size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects in order to force structure sizes to a multiple of 4. Data also have suitable alignment from the beginning of the file. For example, a structure containing `Elf32_Addr` will be aligned on a 4-byte boundary within the file.

For portability, ELF data structures use no bitfields.

ELF Headers

Some ELF file control structures can grow because the ELF header contains their actual sizes. If the ELF file format changes, a program may encounter control structures that are larger or smaller than expected. An ELF header is set by the following C structure declaration:

```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half    e_type;
    Elf32_Half    e_machine;
    Elf32_Word    e_version;
    Elf32_Addr    e_entry;
    Elf32_Off     e_phoff;
    Elf32_Off     e_shoff;
    Elf32_Word    e_flags;
    Elf32_Half    e_ehsize;
    Elf32_Half    e_phentsize;
    Elf32_Half    e_phnum;
    Elf32_Half    e_shentsize;
    Elf32_Half    e_shnum;
    Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

The fields of struct `Elf32_Ehdr` are as follows.

<code>e_ident</code>
Identifies the file as an ELF file and provides machine-independent data to decode the contents. For more information, see “ELF Identification” on page 807.
<code>e_type</code>
Identifies the ELF file type. Permitted values are as follows:
• 0 (<code>ET_NONE</code>): No file type • 1 (<code>ET_REL</code>): Relocatable file • 2 (<code>ET_EXEC</code>): Executable file • 0xFF00 (<code>ET_LOPROC</code>): Begin processor-specific range • 0xFFFF (<code>ET_HIPROC</code>): End processor-specific range

e_machine

Specifies the target architecture for an individual file. Permitted values are as follows:

- 0 (EM_NONE): Invalid machine
- 2 (EM_SPARC): Sun SPARC
- 3 (EM_386): Intel 80386
- 4 (EM_68K): Motorola 68000
- 6 (EM_486): Intel 80486
- 8 (EM_MIPS): MIPS
- 19 (EM_960): Intel i960
- 20 (EM_PPC): PowerPC
- 36 (EM_V800): Renesas V800 series
- 37 (EM_FR20): Fujitsu FR
- 40 (EM_ARM): ARM
- 42 (EM_SH): Hitachi SH
- 44 (EM_TRICORE): Infineon TriCore
- 52 (EM_COLDFIRE): Motorola ColdFire
- 54 (EM_MCORE): Motorola MCore
- 58 (EM_STARCORE): Motorola StarCore
- 60 (EM_ST100): STMicroelectronics ST100
- 78 (EM_FIREPATH): Broadcom FirePath
- 88 (EM_M32R): Renesas M32R
- 106 (EM_BLACKFIN): ADI Blackfin

e_version

Identifies the ELF file version. Permitted values are as follows:

- 0 (EV_NONE): Invalid version
- 1 (EV_CURRENT): Current version

The value 1 (one) signifies the original file format; extensions will create new versions with higher numbers.

e_entry

Specifies the virtual address to which the system first transfers control upon starting the process. If the file has no associated entry point, this field contains the value zero.

e_phoff	Specifies the program header table's file offset in bytes. If the file has no program header table, this field contains the value zero.
e_shoff	Specifies the section header table's file offset in bytes. If the file has no section header table, this field contains the value zero.
e_flags	Specifies processor-specific flags associated with the file. Flag names take the form <i>EF_machine_flag</i> . • 0x80000000 (<i>EF_PPC_EMB</i>): Power PC embedded
e_ehsize	Specifies the ELF header size in bytes.
e_phentsize	Specifies the size in bytes of one entry in the file's program header table; all entries are the same size.
e_phnum	Specifies the number of entries in the program header table. The product of <i>e_phentsize</i> and <i>e_phnum</i> specifies the table's size in bytes. If a file has no program header table, <i>e_phnum</i> contains the value zero.
e_shentsize	Specifies the section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.
e_shnum	Specifies the number of entries in the section header table. The product of <i>e_shentsize</i> and <i>e_shnum</i> specifies the section header table's size in bytes. If a file has no section header table, this field contains the value zero.
e_shstrndx	Specifies the index of the section containing the string table used for section names. By convention, this is the section <i>.shstrtab</i> . If the file has no section name string table, this field contains the value <i>SHN_UNDEF</i> .

ELF Identification

ELF provides a file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header correspond to the `e_ident` member. The identification indexes are tabulated below:

Index name	Value	Meaning
EI_MAG0 to EI_MAG3	0 to 3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_PAD	7	Start of padding bytes
EI_NIDENT	16	Size of <code>e_ident</code> []

These indexes access bytes that contain the following values:

EI_MAG0 to EI_MAG3

A file's first 4 bytes identify the file as ELF as follows:

- `e_ident[EI_MAG0]`: 0x7F (ELFMAG0)
- `e_ident[EI_MAG1]`: 'E' (ELFMAG1)
- `e_ident[EI_MAG2]`: 'L' (ELFMAG2)
- `e_ident[EI_MAG3]`: 'F' (ELFMAG3)

EI_CLASS

This byte identifies the file's class, or capacity. The file format is portable among machines of various sizes; it does not impose the size of the largest machine on the smallest. Permitted values are as follows:

- 0 (ELFCLASSNONE): Invalid class
- 1 (ELFCLASS32): 32-bit objects. Supports machines with files and virtual address spaces up to 4 Gigabytes. It uses the basic types defined above.
- 2 (ELFCLASS64): 64-bit objects. Reserved for 64-bit architectures. Its appearance here shows how the file format may change, but the 64-bit format is otherwise unspecified. Other classes are defined as necessary, with different basic types and sizes for file data.

EI_DATA

This byte specifies the data encoding of the processor-specific data in the file.

Permitted values are as follows:

- 0 (ELFDATANONE): Invalid data encoding
- 1 (ELFDATA2LSB): Specifies 2's complement values, with the least significant byte occupying the lowest address.
- 2 (ELFDATA2MSB): Specifies 2's complement values, with the most significant byte occupying the lowest address.

Other values are reserved and are assigned to new encodings as necessary.

EI_VERSION

This byte specifies the ELF header version number. Currently, this value must be EV_CURRENT, as explained above for e_version.

EI_PAD

This value marks the beginning of the unused bytes in e_ident. These bytes are reserved and set to zero; programs that read ELF files should ignore them. The value of EI_PAD changes if currently unused bytes are given meanings.

EI_NIDENT

This value specifies the size of e_ident[].

ELF Sections

Section Headers

You can use an ELF file's section header table to locate all the file's sections. The section header table is an array of **Elf32_Shdr** structures. A section header table index is a subscript into this array. The ELF header e_shoff specifies the byte offset from the beginning of the file to the section header table; e_shnum tells how many entries the section header table contains; and e_shentsize specifies the size in bytes of each entry.

Sections contain all information in a file except the ELF header, the program header table, and the section header table. Sections satisfy several conditions:

- Every section in an ELF file has exactly one section header describing it. Section headers may exist that do not have an associated section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.

- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An ELF file may have inactive space. The various headers and the sections might not account for every byte in a file. The contents of the inactive space are unspecified.

A section header has the following structure:

```
typedef struct {
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off  sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
} Elf32_Shdr;
```

The fields of **struct Elf32_Shdr** are as follows.

sh_name	Specifies the name of the section. Its value is an index into the section header string table section and specifies the location of a null-terminated string.
sh_type	Specifies the section's contents and semantics. For more information, see "Section Attribute Flags" on page 811.
sh_flags	Specifies 1-bit flags that describe miscellaneous attributes of the section.
sh_addr	Specifies the address at which the section's first byte should reside if the section appears in the memory image of a process. Otherwise, this field contains zero.
sh_offset	Specifies the byte offset from the beginning of the file to the first byte in the section.
sh_size	Specifies the section's size in bytes. Unless the section type is <code>SHT_NOBITS</code> , the section occupies <code>sh_size</code> bytes in the file. A section of type <code>SHT_NOBITS</code> may have a non-zero size, but it occupies no space in the file.

sh_link
Contains a section header table index link, whose interpretation depends on the section type.
sh_info
Contains extra information, whose interpretation depends on the section type.
sh_addralign
Some sections have address alignment constraints. For example, if a section contains a doubleword, the system may need to ensure doubleword alignment for the entire section. That is, the value of sh_addr must be equal to 0 (zero), modulo the value of sh_addralign. Currently, only 0 (zero) and positive integral powers of two are allowed. The values 0 (zero) and 1 (one) mean the section has no alignment constraints.
sh_entsize
Some sections contain a table of fixed-size entries, such as a symbol table. For such sections, this field specifies the size in bytes of each entry. This structure contains zero if the section does not contain a table of fixed-sized entries.

Special Section Indexes

Symbol table entries index the section table through the st_shndx field. See “Symbol Tables” on page 818.

Name	Value	Meaning
SHN_UNDEF	0	A meaningless section. A symbol “defined” relative to this section is an undefined symbol.
SHN_COMMON	-14	Common, or .bss, symbols are allocated space in this section.
SHN_ABS	-15	Contents of the section are absolute values; they are not affected by relocation.
SHN_GHS_SMALLCOMMON	-256	Not available for all processors. Similar to SHN_COMMON, but for a limited number of small variables allocated to SDA.
SHN_GHS_ZERO_COMMON	-255	Not available for all processors. Similar to SHN_COMMON, but for a limited number of small variables allocated to ZDA.
SHN_GHS_TINYCOMMON	-254	Not available for all processors. Similar to SHN_COMMON, but for a limited number of small variables allocated to TDA.



Note Although index 0 is reserved as an undefined value, the section header table does contain an entry for it.

Section Types

A section header's `sh_type` specifies the section's semantics as follows:

Name	Value	Meaning
<code>SHT_NULL</code>	0	Marks the section header as inactive. It does not have an associated section. Other members of the section header have undefined values.
<code>SHT_PROGBITS</code>	1	Contains information defined by the program that created the ELF file, whose format and meaning are determined solely by the program.
<code>SHT_SYMTAB</code>	2	Contains a symbol table. An object file may have only one section of this type, but this restriction may be relaxed in the future. The table provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking.
<code>SHT_STRTAB</code>	3	Contains a string table. A file may have multiple string table sections.
<code>SHT_RELA</code>	4	Contains relocation entries with explicit addends, such as type <code>Elf32_Rela</code> for the 32-bit class of ELF files. A file may have multiple relocation sections.
<code>SHT_NOTE</code>	7	Contains notes that flag this ELF file indicating certain properties.
<code>SHT_NOBITS</code>	8	Occupies no space in the file but otherwise resembles <code>SHT_PROGBITS</code> .
<code>SHT_REL</code>	9	Contains relocation entries without explicit addends, such as type <code>Elf32_Rel</code> for the 32-bit class of ELF files. A file may have multiple relocation sections.

Section Attribute Flags

A section header's `sh_flags` field contains 1-bit flags that describe the section's attributes.

Name	Value	Meaning
SHF_WRITE	0x1	Contains data that should be writable during process execution.
SHF_ALLOC	0x2	Occupies memory during process execution. Some control sections do not reside in the memory image of the final application; this attribute is off for those sections.
SHF_EXECINSTR	0x4	Contains execution machine instructions.
SHF_GHS_ABS	0x400	Has an absolute, non-relocatable address.

Two structures in the section header, `sh_link` and `sh_info`, contain special information, depending on section type as follows:

<code>sh_type</code>	<code>sh_link</code>	<code>sh_info</code>
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding <code>STB_LOCAL</code>).
<i>other</i>	SHN_UNDEF	0

Section Names

Section names beginning with a period (.) are not meant for general use by application programs, although application programs may use these sections if their existing meanings are satisfactory. Application programs may use names without the leading period to avoid conflicts with predefined section names.

Frequently Used Sections

Some sections contain program and control information. Sections in the following list have predefined meanings, with the indicated types and attributes.

<code>.bss</code> Contains uninitialized data that contributes to the program's memory image. The system initializes the data with zeros when the program begins to run. Type: <code>SHT_NOBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_WRITE</code>
<code>.data</code> Contains initialized data that contributes to the program's memory image. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_WRITE</code>
<code>.linfix</code> When using Code Factoring or fixing "out of range relocations" inline, the assembly becomes out of sync with the debugging information. The linker notes the changes to the program that require updates to the debugging information in the <code>.linfix</code> section. MULTI merges the information in the <code>.dla</code> file and the <code>.linfix</code> section when debugging the program. The <code>.linfix</code> section is not downloaded to the target. Type: <code>SHT_PROGBITS</code> Attributes: None
<code>.relname</code> <code>.relaname</code> These sections contain relocation information. Conventionally, <i>name</i> is supplied by the section to which the relocations apply. A relocation section for <code>.text</code> has the name <code>.rel.text</code> or <code>.rela.text</code>. Types: <code>SHT_REL</code> and <code>SHT_RELA</code> Attributes: none
<code>.rodata</code> Read-only data area. Similar to the <code>.data</code> section, but composed of constant data. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code>
<code>.shstrtab</code> Contains section names. Type: <code>SHT_STRTAB</code> Attributes: None

<code>.strtab</code> Contains strings, most commonly the strings that represent the names associated with symbol table entries. Type: <code>SHT_STRTAB</code> Attributes: none
<code>.symtab</code> Contains a symbol table. Type: <code>SHT_SYMTAB</code> Attributes: none
<code>.text</code> Contains the “text”, or executable instructions, of a program. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_EXECINSTR</code>

Some sections are specific to Green Hills ELF; these are listed in the following table. Target processors may support some or all of these sections.

<code>.syscall</code> A program code section to support the Green Hills system call mechanism. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_EXECINSTR</code> <code>SHF_ALLOC</code>
<code>.secinfo</code> A table, created by the linker, that describes actions to be taken on sections as they are loaded for program execution (sections to be cleared, copied from ROM to RAM, and so on). Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code>
<code>.fixaddr</code> <code>.fixtype</code> Tables created by the compiler for position independent code (PIC) and position independent data (PID) static pointer adjustments to be made when the program is loaded for execution. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code>

<code>.sdabase</code> If not in PID mode, the run-time system initializes the Small Data Area (SDA) base register to be the address of this section (<code>.sdabase</code>). Type: <code>SHT_NULL</code> Attributes: <code>SHF_ALLOC</code>
<code>.sdata</code> Initialized Small Data Area <code>.zdata</code> Initialized Zero Data Area. Similar to the <code>.data</code> section, but of limited size and more efficiently addressed. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_WRITE</code>
<code>.sbss</code> Uninitialized Small Data Area. <code>.PPC.EMB.sbss0</code> Uninitialized Zero Data Area. Similar to the <code>.bss</code> section, but of limited size and more efficiently addressed. Type: <code>SHT_PROGBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_WRITE</code>
<code>.heap</code> Section describing the area of memory for dynamic allocations through malloc and related functions. Type: <code>SHT_NOBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_WRITE</code>
<code>.stack</code> Section describing the area of memory that the program stack will occupy. Type: <code>SHT_NOBITS</code> Attributes: <code>SHF_ALLOC</code> <code>SHF_WRITE</code>

Relocation Types

Relocation entries contain this information. There are two forms of relocation entries, corresponding to `SHT_REL` and `SHT_RELA` sections. Relocations

can have offsets called addends from the symbols being referenced by the `ELF32_R_SYM()` information in `r_info`. In `SHT_REL`, the addends are encoded in the instruction or data fields of the location referenced by the `r_offset` field. In `SHT_RELA`, each relocation entry contains a dedicated `r_addend` field. The `SHT_RELA` format is preferred because it simplifies the interpretation of relocations, as well as allows offsets that are greater than the fields can contain.

Example 1. A Relocation Entry Corresponding to `SHT_REL`

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
} Elf32_Rel;
```

Example 2. A Relocation Entry Corresponding to `SHT_RELA`

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
    Elf32_Sword r_addend;
} Elf32_Rela;
```

`r_offset`

Specifies the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value of which is the virtual address of the storage unit affected by the relocation.

`r_info`

Specifies both the symbol table index with respect to which the relocation must be made and the type of relocation to apply information for a process's program image. For example, a call instruction's relocation entry contains the symbol table index of the function being called. You may find the following macros helpful when reading from or writing to the `r_info` field:

```
#define ELF32_R_SYM(i)    ((i) >> 8)
#define ELF32_R_TYPE(i)  ((unsigned char) (i))
#define ELF32_R_INFO(s,t) (((s)<<8) + (unsigned char) (t))
```

`r_addend`

Specifies a constant value used to compute the final value to be stored into the relocatable field.

A relocation section references two other sections: a symbol table and a section to modify. The section header's `sh_link` and `sh_info` specify these sections.

Symbol Tables

The symbol table of an ELF file contains information to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 (zero) both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified below. A symbol table entry has the following format:

```
typedef struct {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
} Elf32_Sym;
```

where:

st_name	Contains an index into the file's symbol string table. The string table contains the character representations of the symbol names. If this field is non-zero, then it represents a string table index that specifies the symbol name. Otherwise, the symbol table entry has no name.
st_value	Specifies the associated symbol's value. Depending on the context, this may be an absolute value, an address, and so on.
st_size	Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This field is zero if the symbol has no size or an unknown size.
st_info	Specifies the symbol's binding attributes and type, as explained in the symbol binding and symbol type sections below. The values and meanings are defined below: <pre>#define ELF32_ST_BIND(i) ((i)>>4) #define ELF32_ST_TYPE(i) ((i)&0xf) #define ELF32_ST_INFO(b,t) (((b)<<4)+(t)&0xf))</pre>

st_other
Contains the value zero and has no defined meaning.
st_shndx
Every symbol table entry is “defined” in relation to some section; this field contains the relevant section header table index. Some section indexes indicate special meanings. For more information, see “Special Section Indexes” on page 810.

Symbol Binding

A symbol’s binding determines the linkage visibility and behavior.

Name	Value	Attributes
STB_LOCAL	0	Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.
STB_GLOBAL	1	Global symbols are visible to all object files being combined. One file’s definition of a global symbol will satisfy another file’s undefined reference to the same global symbol.
STB_WEAK	2	Weak symbols resemble global symbols, but their definitions have lower precedence.
STB_LOPROC	13	Values in this inclusive range (from STB_LOPROC to STB_HIPROC) are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.
STB_HIPROC	15	

Symbol Type

A symbol’s type provides a general classification for the associated entity.

Name	Value	Attributes
STT_NOTYPE	0	The symbol’s type is not specified.
STT_OBJECT	1	The symbol is associated with a data object, such as a variable, array, and so on.
STT_FUNC	2	The symbol is associated with a function or other executable.

Name	Value	Attributes
STT_SECTION	3	The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.
STT_FILE	4	By convention, this symbol names the source file associated with the object file. For this symbol, the binding is STB_LOCAL and the section index is SHN_ABS. All STT_FILE symbols precede any other STB_LOCAL symbols.
STT_LOPROC	13	Values in this inclusive range (from STT_LOPROC to STT_HIPROC) are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.
STT_HIPROC	15	

Symbol Values

Symbol table entries for different ELF file types have slightly different interpretations for the `st_value` member:

- In relocatable files, `st_value` contains alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, `st_value` contains a section offset for a defined symbol. That is, `st_value` is an offset from the beginning of the section that `st_shndx` identifies.
- In executable files, `st_value` contains a virtual address. To make these symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different file types, the information allows efficient access by the appropriate programs.

String Tables

String table sections contain null-terminated character sequences or strings. These strings are used to represent symbol and section names. An empty string table section is permitted; its section header's `sh_size` contains zero. Non-zero indexes are invalid for an empty string table. If the string table is not empty, you can reference a string as an index into the string table section. The first byte, which is index 0 (zero), contains a null character. Likewise, a string table's last byte contains a null character to ensure null termination for all strings. A string whose index is 0 (zero) specifies either no name or a null name, depending on the context.

A section header's `sh_name` contains an index into the section header string table (See the `e_shstrndx` field in “ELF Headers” on page 804). The following figures show a string table with 25 bytes and the strings associated with various indexes:

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	v	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

The string table indexes are:

Index	String
0	<i>none</i>
1	name.
7	Variable
11	able
16	able
24	<i>null string</i>

A string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist, and a single string may be referenced multiple times. Strings that are not referenced are also allowed.

Program Headers

An ELF executable file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. A file segment contains one or more sections. Program headers are defined only for executable files. A file specifies its own program header size with the ELF header's `e_phentsize` and `e_phnum`.

```
typedef struct {
    Elf32_Word p_type;
    Elf32_Off  p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
} Elf32_Phdr;
```

`p_type`

The kind of segment this array element describes and how to interpret the array element's information. For more information, see "Program Types" on page 823.

`p_offset`

Offset from the beginning of the file at which the first byte of the segment resides.

`p_vaddr`

Address at which the first byte of the segment resides in memory.

`p_paddr`

For targets where physical addressing is relevant, this field gives the segment's physical address. For other targets this field is unused and is set to zero by the linker.

`p_filesz`

Number of bytes in the file image of the segment; it may be zero.

`p_memsz`

Number of bytes in the memory image of the segment; it may be zero.

`p_flags`

Flags relevant to the segment. For more information, see “Program Attribute Flags” on page 824.

`p_align`

Loadable process segments must have congruent values for `p_vaddr` and `p_offset`, modulo the page size. This field specifies the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, `p_align` should be a positive, integral power of 2, and `p_vaddr` should equal `p_offset`, modulo `p_align`.

Program Types

A program header’s `p_type` specifies the program’s semantics as follows:

Name	Value	Meaning
PT_NULL	0	The array element is unused; other member values are undefined. This type lets the program header table have ignored entries.
PT_LOAD	1	The array element specifies a loadable segment, described by <code>p_filesz</code> and <code>p_memsz</code> . The bytes from the file are mapped to the beginning of the memory segment. If the segment’s memory size (<code>p_memsz</code>) is larger than the file size (<code>p_filesz</code>), the “extra” bytes are defined to hold the value 0 and to follow the segment’s initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the <code>p_vaddr</code> member.
PT_DYNAMIC	2	The array element specifies dynamic linking information.
PT_INTERP	3	The array element specifies the location and size of a null-terminated pathname to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry.
PT_NOTE	4	The array element specifies the location and size of auxiliary information.

Name	Value	Meaning
PT_SHLIB	5	This segment type is reserved but has unspecified semantics.
PT_PHDR	6	The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry.
PT_LOPROC to PT_HIPROC	0x70000000 to 0x7fffffff	Reserved for processor-specific semantics.

Program Attribute Flags

A program to be loaded by the system must have at least one loadable segment (although this is not required by the file format). When the system creates loadable segment memory images, it gives access permissions as specified in the `p_flags` member. All bits included in the `PF_MASKPROC` mask are reserved for processor-specific semantics.

Name	Value	Meaning
PF_X	0x1	Execute
PF_W	0x2	Write
PF_R	0x4	Read
PF_MASKPROC	0xf0000000	Unspecified

If a permission bit is 0, that type of access is denied. Actual memory permissions depend on the memory management unit, which may vary from one system to another. Although all flag combinations are valid, the system may grant more access than requested. However, a segment will never have write permission unless it is specified explicitly. The following table shows both the exact flag interpretation and the allowable flag interpretation.

Flag	Value	Exact	Allowable
(none)	0	All access denied	All access denied
PF_X	1	Execute only	Read, execute
PF_W	2	Write only	Read, write, execute
PF_W+PF_X	3	Write, execute	Read, write, execute
PF_R	4	Read only	Read, execute
PF_R+PF_X	5	Read, execute	Read, execute
PF_R+PF_W	6	Read, write	Read, write, execute
PF_R+PF_W+PF_X	7	Read, write, execute	Read, write, execute

For example, typical text segments have read and execute - but not write - permissions. Data segments normally have read, write, and execute permissions.

Appendices

Part IV

GNU Options

Appendix A

The builder currently accepts the following legacy GNU option names as aliases for official Green Hills options. These option aliases are deprecated. We recommend that you use the official Green Hills Software options.

Legacy GNU Option	Official Green Hills Option
###	#
-EB	-bigendian
-EL	-littleendian
-maltivec	-AltiVec
-march=	-cpu=
-mcpu	-cpu=
-mlong-calls	-farcalls
-mno-altivec	-noAltiVec
-mno-long-calls	-nofarcalls
-msdata=none	-sda=0



Note

1. Of the options listed in the previous table, only **-EB**, **-EL**, **-mlong-calls**, and **-mno-long-calls** are recognized by the Builder. The remaining options are recognized only by the driver.

Green Hills Project File Format

Appendix B

This Appendix Contains:

- Green Hills Project File Syntax
- Top Project Directives
- Conditional Control Statements
- Macro Functions

Green Hills project (**.gpj**) files support several advanced methods for specifying the files and options to use when building your project with the MULTI Project Manager or command line build utility. This appendix describes the file format and covers advanced features that you can use only by manually editing project files.

Green Hills Project File Syntax

Green Hills project (**.gpj**) files have three sections:

- Header section — Specifies the project file's type. For Top Project files, also specifies project-wide directives.
- Options section — Specifies options that apply to the project and all of its children.
- Children section — Specifies files included by the project, their types, and options that apply to each respective file.

In any of these sections:

- You can add comments anywhere after the first line of a project file with the hash symbol (`#`). Comments continue to the end of the line.
- You can use the value of a macro or imported environment variable by preceding the name of the macro or variable with the dollar sign (`$`). For example, to use the value of the macro `my_library` as part of the `-l` option, type the following:

```
-l$my_library
```

The Header Section

The header is the first section in a project file. The first line in a project file (and therefore the first line of the header) must be `#!gbuild`. The last line in the header is the project type, enclosed in brackets (`[]`). For a list of types, see the Project Files (**.gpj**) list in “File Types” on page 39.

For example, a header for a simple program project might look like this:

```
#!gbuild
# My Hello World Program
[Program]
```

Top Project headers must specify a `primaryTarget`, and may specify any number of additional Top Project directives. For more information about these directives, see “Top Project Directives” on page 835. For example:

```
#!/gbuild
# My Top Project
# Additional Top Project directives go here
primaryTarget=ppc_standalone.tgt
[Project]
```

The Project Options Section

The project options section lists options that apply to the project you are editing and all of its children. Each line must be indented by at least one whitespace character. You can put multiple options on the same line, separated by white spaces. If the option value contains spaces, enclose it in double quotes (" "). You may need to escape backslashes inside quoted strings. The following example builds on the previous Top Project example:

```
#!/gbuild
# Header section
primaryTarget=ppc_standalone.tgt
[Project]

# Options section
    -bsp generic           # No specified BSP
    -G                     # Generate debug information
    -object_dir="my objs" # Put object files in ./my objs
    -DDEBUG -DNOISY       # Two preprocessor defines
```

The Children Section

The children section lists all files that are children of the project you are editing. The filenames are not indented, and may use full or relative paths. If you use a relative path, the Project Manager tries to resolve the file relative to the current project file’s directory, and then relative to each directory specified by applicable `:sourceDir` options (see “**Source Directories Relative to This File**” on page 174). If the filename contains spaces, enclose it in double quotes (" "). You may need to escape backslashes inside quoted strings.



Tip If you plan to open your project on hosts that run different operating systems, always use forward slashes (/) when specifying paths or filenames.

Each line that specifies a child file must specify the file's type in brackets ('[]'), unless it is the default type for that file's extension. For a list of defaults, see "File Types" on page 39. If you do not specify a type and there is no default, it is treated as a text (.txt) file and is not included in the build. For example:

```
# Children section
src_hello\hello.gpj    [Program] # Override default [Subproject]
tgt\resources.gpj      [Project] # Override default [Subproject]
docs\readme_one.txt    # Extension sets type to [Text]
docs\readme_two.odd    # No default, treated as [Text]
```

The type you specify for any child project must match the type in that project's header. For example, if your Top Project includes a child program:

```
#!gbuild
# My Top Project default.gpj
primaryTarget=ppc_standalone.tgt
[Project]
src_hello\hello.gpj      [Program] # The child program
```

The child must have [Program] as the file type in its header:

```
#!gbuild
# My Program src_hello\hello.gpj
[Program]                # The type specified in the parent
```


Any indented, non-commented lines immediately following a child entry are treated as a child options section, which lists options that apply to that child and all of its children. These options do not apply to the project file you are editing or any of the child's siblings. Child option sections have the same syntax as the project options section. For example, if you used the following project file, **foo.o** would be placed in **objs\special**, while **hello.o** would be placed in **objs\hello**:

```
#!/gbuild
[Program]

# Options Section
    -object_dir=objs\hello

# Children Section
foo.c
    # Child options for foo.c
    -object_dir=objs\special
hello.c
tgt\standalone_ram.ld
```

Top Project Directives

Top Project directives are directives that specify settings for your entire project. You can only use them in your Top Project. There are two groups of these directives:

- Group 1 directives
- Group 2 directives

All group 1 directives must appear before group 2 directives, and all Top Project directives must appear before the project type line (usually [Project]). To add a Top Project directive:

1. Open the MULTI Project Manager.
2. Right-click your Top Project and select **Edit**. The Editor window opens.
3. Type the directive above the line that reads [Project].
4. Save and close the file.
5. To load your changes into the Project Manager, select **File → Reload Saved Project**.

The following tables list available directives.

Group 1 Directives

All of these directives are optional.

defaultBuildOptions= <i>option...</i>
--

Takes a space-separated list of gbuild options and applies them to this project and any subprojects, as if you had passed them on the command line. For example, if you add the line:
--

<pre>defaultBuildOptions=-all</pre>

to your Top Project, every file is rebuilt each time you build your project.
--

For a list of options you can use with this directive, see “The gbuild Utility Program” on page 499.
--

import <i>variable_name</i>

Imports the environment variable named <i>variable_name</i> from the local environment. For information about how to import environment variables using the GUI, see “Importing Environment Variables” on page 61.
--

macro <i>macro_name=value</i>

Defines a new macro with the given name and value. There must not be any spaces between <i>macro_name</i> , <code>=</code> , and <i>value</i> . For information about how to define macros using the GUI, see “Setting Build Macros” on page 60.
--

Group 2 Directives

customization= <i>customization_file</i>

This optional directive specifies a customization (.bod) file for your project. You can use this directive multiple times.

For more information about customization files, see Appendix C.

primaryTarget= <i>target_file</i>
--

This required directive specifies the target file for your project. This is one of the .tgt files included in the MULTI distribution that contains all of the target-specific settings necessary to compile code for your target.
--

Conditional Control Statements

Conditional control statements reside in the project file and are used to indicate the portion of the project tree and options used in the build process. These control statements are especially useful when one portion of the project tree is built in multiple ways (for example, multiple targets). Each statement is evaluated during the build using information from the current file in the project tree and all of its parents. If the statement evaluates to true, the option or file is included in the build. If the statement evaluates to false, the option or file is silently skipped.



Note When an option is modified by a conditional control statement, you cannot use the **Build Options** window to modify that specific option, but you can usually use the **Build Options** window to add additional items to a list option or override the option in a subproject.

Some rules for using conditional control statements are:

- You can only specify these statements by manually editing the **.gpj** files.
- Conditional control statements can be used with options or filenames. The syntax is `{condition}` before the option or filename. For example:

```
{VERSION>5} ver5.c  
    {isdefined(DEBUG)} -D DEBUG
```

- You can specify more than one conditional control statement for a filename. All statements must evaluate to true for the file to be included in the build. For example:

```
{optional}{isdefined(SUPPORTS_DSP)} dsp_subsystem.gpj
```



Note Conditional control statements applied to options must have a whitespace before them in order to function properly.

Conditional Control Statement Syntax

The following are valid conditional control statements:

command=command_name

The option or file is included when building the project if the current file is processed using the specified command.

command_name must be the same as the `command` property specified in the `FileType` section of the `.bod` file (see “Components of Customization Files” on page 844).

comment

The file is never included when building the project, and is disabled in the Project Manager. **comment** applies to source or project files, but not options.

filetype=filetype

The option is included when building the project if the current file is of the specified type. *filetype* must match the `name` property specified in the `FileType` section of the `.bod` file (see “File Type Definitions” on page 845).

filetype=filetype is only useful for options that are inherited by multiple types.

expression

The option or file is included when building the project if the expression evaluates to `true`. The expression can include macros, unsigned integers, the following functions, standard Boolean operators, and standard comparison operators. If you pass a string to one of the following functions and that string begins with a numerical digit, you must put quotes (") around the string.

The following are accepted functions:

- `isdefined(macro_name)` — Evaluates to `true` if *macro_name* is currently defined, and `false` otherwise.
- `isundefined(macro_name)` — Evaluates to `true` if *macro_name* is not currently defined, and `false` otherwise.
- `istarget(target_name)` — Evaluates to `true` if *target_name* is equal to the name of the project target, and `false` otherwise. The string *target_name* corresponds to the base name of the **.tgt** file defined as the `primaryTarget` in your Top Project. The comparison is case-insensitive, and *target_name* must not include the **.tgt** extension when used in this conditional expression. For example, if the target is **ppc_standalone.tgt**, then `istarget("ppc_standalone")` will evaluate to `true`.
- `streq(macro_name, string, ...)` — Evaluates to `true` if the string value of *macro_name* is equal to that of one or more of the provided *string* arguments, and `false` otherwise. The comparison is case-sensitive.
- `strprefix(macro_name, string, ...)` — Evaluates to `true` if one or more of the provided *string* arguments are a prefix of the string value of *macro_name*, and `false` otherwise. The comparison is case-sensitive.

Note: Macros used in a macro expression must be valid C identifiers.

The following are accepted operators:

- Standard Boolean operators — `!`, `&&`, and `||`
- Standard comparison operators — `>=`, `<=`, `>`, `<`, `==` and `!=`

For example:

```
{ isdefined(DEBUG_LEVEL) && DEBUG_LEVEL > 0 } -G
```

nobuild

The file is never included when building the project, but is still included in the Project Manager. **nobuild** applies to source or project files, but not options.

optgroup=option_group

The option or file is included if the command used to process the current file supports the specified option group (see “Components of Customization Files” on page 844).

optional

When applied to a file, the current file is only included when building the project if the file exists.

When applied to an option, the option is only included when building the project if the option is supported by the current target.

pass= [COMPILE] [LINK] [DEPEND] [FIRSTPASS] [ALL]

Causes the controlled option to only be used during the specified processing pass. **pass=** only applies to options.

rebuild

Causes the current file to be rebuilt every time the project is built. **rebuild** only applies to source files (not **.gpj** files or options).

target=target_name

The option or file is included when building the project for the specified target.

Note: This syntax is deprecated. Use `istarget(target_name)` instead.

target!=target_name

The option or file is not included when building the project for the specified target.

Note: This syntax is deprecated. Use `!istarget(target_name)` instead.

Special Conditional Control Statements

The following are special conditional control statements that can only be applied to *Select One* **.gpj** files. These control statements do not selectively include or exclude the project that they are applied to. Instead, they control the printing of warning and errors when processing Select One project files.

selectone_optional

If an assembly file corresponding to the current target or a C file is not specified, the project is silently ignored. This behavior is similar to the effect of the **{optional}** condition when applied to files.

selectone_required

If an assembly file corresponding to the current target or a C file is not specified, an error is generated, which stops the build.

Macro Functions

Macro functions allow you to define a macro that does not have a static value. Instead, the value is determined by evaluating a function. You can place them wherever you could place a standard macro. To place a macro function, use the following syntax:

```
${macro_function}
```

Macro functions cannot be nested inside one another, but can contain references to standard macros. The available macro functions are:

```
%expand_path(relative_path)
```

Expands *relative_path* to an absolute path. *relative_path* is resolved relative to the directory that contains the Top Project. For example:

```
macro SCRIPTS=${%expand_path(../scripts)}
```

```
%option_value(option_name)
```

Given a name of an option, this function returns the option's set value. This function only supports options that take string arguments.

option_name must match the `name` parameter of the option as defined in the **.bod** file, and must include the dash or other punctuation that normally accompanies the option name (for example, `-os_dir`). Do not quote the option name.

This function searches the entire set of options that apply to the build item in which it is used. If the specified option is not set, or if the option does not take a string argument, this function returns an empty string and issues a warning.

For example:

```
-DTHE_OS_DIR=${%option_value(-os_dir)}
```

For more information about macros, see “Setting Build Macros” on page 60.

Customization Files

Appendix C

This Appendix Contains:

- Components of Customization Files
- Adding a Customization File to a Project
- Extended Examples
- Customization File Syntax

Components of Customization Files

Customization files, or **.bod** files, are used to customize your build environment by defining acceptable input file types and how to build them. Customization files contain three types of definitions:

- **FileTypes** — Definitions of types of files that can be built or included in project files.
- **Commands** — Definitions of the commands used to build or edit the various file types.
- **CommandOptions** — Definitions of the options supported by the commands defined in the **Commands** section. These definitions are broken into option groups, where each group represents the options supported by one or more commands.

The following sections describe the purpose of each type of definition and its syntax by using the **Custom File Types** demo project available from the Project Manager. For information about creating a project including this example, see “Creating A New Project” on page 22.



Note If you are writing a customization file for use with both Windows and Linux/Solaris hosts, we recommend that you use forward slashes (/) when specifying any pathnames.



Note The Green Hills tools use **.bod** files to define the file types, commands, and options available by default. These **.bod** files are a useful resource as you create your own customization file. The **.bod** files can be found in the **defaults/bld_rules** subdirectory of your MULTI installation. The file **ghs_filetypes.bod** contains file type and command definitions. The files **ghs_*_options.bod** contain the command option definitions. Use these files for reference only; do not modify them.

Customization File Definitions

The following section will provide you with definitions used in **.bod** files, as well as examples for creating a customized file.

File Type Definitions

The FileTypes section of a customization file defines additional types of files that can be built as part of your project. The definitions in this section provide the Green Hills tools with information about how to identify and process these files.

```
FileTypes {
# FileTypes defines our custom types and how to
# fill in the command arguments

  BinInput {
# Here we've created a new custom type called "Binary Input"
    name="Binary Input"
    extensions={"bin"} # The input extensions are .bin.
    # It outputs .c files (used for -clean).
    outputExtension="c"
    # It outputs source files instead of object files
    outputType="SourceFile"
    # Do not perform grep operations on this input.
    grepable=false
    # Use the matching command from the Commands section below.
    command="GHS Binary Converter"
    # Specifies the exact command line using
    # variables to substitute for context-sensitive elements.
    commandLine="$COMMAND $OPTIONS $INPUTFILE -o $OUTPUTFILE"
    # What to show in the progress/details windows.
    progress="Converting"
    # What to show in the right click menu.
    action("&Convert"
    color="#500050" # Color for files of this type.
    # Pass this file onto the link line (see below).
    findLinkLine=true
    # Run during the first pass on multi-pass builds.
    promoteToFirstPass=true
  }

# The following two lines describe how to pass this file
# on the link line.

# Here we're passing this as an extra_file, which adds a
# reference to the file into the debug info
# (so that you can "e" to the .bin file).
Program.linkableTypes += {"Binary Input"}
Program.linkSpecifiers += {"Binary Input;-extra_file=$(INPUTFILE)"}
}
```

Each file type definition is contained within curly braces, can have an optional identifier (BinInput in the preceding example), and contains numerous properties.

The only required property is `name`, which is a string used to identify the type in the Project Manager. There are also a few other properties that are necessary for the build tools to automatically recognize and build the custom file type.

You can add the `extensions` property, which is a list of file extensions that automatically imply custom file types. Add the `command`, `commandLine` and `outputType` properties to specify how to build the custom file type. See “Customization File Syntax” on page 856 for additional details about these and other properties.

If your custom input file is used during a build, you can add that file to the debugging information generated for the final executable. By adding the file to the debugging information, you can access it through the MULTI Debugger by using the **e filename** command.

The preceding example demonstrates this through the use of two lines that append values to the `Program.linkSpecifiers` and `Program.linkableTypes` properties. These lines modify properties of the Program file type, which is a standard file type defined in **ghs_filetypes.bod**.



Note `Program.linkSpecifiers` and `Program.linkableTypes` are not included in the curly braces surrounding the rest of the Binary Input file type definition.

- `Program.linkableTypes` — Lists the names of file types that are included on the linker command line when building the parent Program.
- `Program.linkSpecifiers` — Lists the file type names and the corresponding option to include on the linker command line when building the parent program.

The **-extra_file** option indicates that the `INPUTFILE` in the previous example should be added to the debugging information and not to the linked executable. If you add a `FileType` value to the `linkableTypes` property but not to the `linkSpecifiers` property, the output file is added to the linked executable.

Command Definitions

The `Commands` section of a customization file defines programs or commands required to build or edit custom file types. The definitions specify where the programs are found and which groups of options are supported.

```
Commands {  
#Commands contains a list of custom commands and how to locate them  
  GHS_Binary_Converter {  
    # name of the command referred to in FileType  
    name="GHS Binary Converter"  
    # GHS_TOOLS_DIR is a macro specifying the location of "gbuild"  
    exec="${GHS_TOOLS_DIR}/gbin2c"  
    # option sets used by this command  
    options={"SpecialOptions", "GBCOptions"}  
  }  
}
```

Each command definition is contained within curly braces, can have an optional identifier (in this example, `GHS_Binary_Converter`), and contains several required properties. The required properties are:

- `exec` — The name of the program to run. If it contains spaces, the Builder treats the entire string as the program name (it does not parse the string into arguments). If the program name is specified without a directory, the builder searches for the program in the directories of the `PATH` environment variable.
- `name` — A string used to identify the command in `FileType` definitions.
- `options` — A list of option groups supported by this command. All commands must list `SpecialOptions` because this option group is supported by the Builder rather than the executed command.

For more information about other command definition properties, see “Customization File Syntax” on page 856.

CommandOption Definitions

The `CommandOptions` section of a customization file defines groups of command line options supported by the custom commands. Most commands should have their own option group.

```
CommandOptions {
  GBCOptions { # -- Option Set Name --
    # options for the GHS Binary Converter (referred to in the
    # Commands section)

    SizeOpt {
      # Size option for determining whether to output lengths

      {
        name="-size"    #exact name of the option passed
        value=0         #a unique identifier
        enumLabel="On"  #text to show when selecting this option
      }

      {
        #Fake options can be added so that a real option can be disabled
        #lower in the hierarchy even though the tool has no such option.
        #When the command is invoked, no option will be passed.

        name="-no_size"    #option name (stored in .gpj files)
        value=1            #unique value for this option
        enumLabel="Off"    #text to show when selecting this option
        flags={"FAKEOPTION"} #don't pass anything on the command line
      }

      delimiter="NoArg"  #this is a stand-alone option (no arguments)
      merge="Replace"    #each occurrence in the hierarchy
                        #replaces those above it
      optionType="Enum"  #this is an enumerated value option
      guiLabel="Output the length of each array"
                        #Text to display in the options dialog
      guiCategory="Binary Converter"
                        #Category to add this option into
    }
  }
}
```

Each option group is contained within curly braces, must have an identifier (GBCOptions in the example), and includes one or more option definitions. An option definition is also contained within curly braces, can have an identifier (SizeOpt in the example), and specifies properties used by the Builder when constructing command lines.

All option definitions must have at least one name, a delimiter, a merge rule, and a optionType. The name of the option is the exact string to pass to the command. For options that take a parameter, the name is the prefix (for example, **-option=param** has the name “-option”). The delimiter specifies the connector between the option and the parameter (“Equal” for **-option=param**). The merge rule specifies how multiple instances of the

option, on different or the same levels of the hierarchy, are handled. The `optionType` determines which merge rules are available.

An option definition will contain more than one name if the option has more than one valid spelling, or if it has a fixed set of choices instead of a free-form parameter. In either of these scenarios, create a sub-definition by adding curly braces around each name and its corresponding value. Options with the same `value` setting are all considered synonyms. The first option listed is treated as the canonical spelling and is used when the option is set from the Project Manager. For more information, see “Customization File Syntax” on page 856.

Adding a Customization File to a Project

To add a customization file to your project:

1. Select **Edit** → **Set Build Target** to open the **Target Selector** window.
2. Click the **Customizations** button.
3. Click the **Add** button in the **Build Customization** dialog box.
4. In the file chooser that appears, select a custom **.bod** file to add.

Extended Examples

The following sections provide additional reference examples to use when creating customization files.

Example 1. Using Scripts to Process Input Files

The subsequent example demonstrates how to use Perl scripts to convert input files into source code. This example defines a single command for the Perl interpreter, which is used to run scripts for four different custom file types.

```
%define scriptdir c:/scripts

Commands {
{
    name="Perl Script"
    exec="perl"
    options={"SpecialOptions"}
}
}

FileTypes {
{
    name="Prtcl"
    extensions={"prtcl"}
    outputType="SourceFile"
    grepable=true
    extraFiles+={"$(OUTPUTDIR)/$(OUTPUTNAMEBASE)_gen.c"}
    extraFiles+={"$(OUTPUTDIR)/$(OUTPUTNAMEBASE)_gen.h"}
    extraFiles+={"$(OUTPUTDIR)/$(OUTPUTNAMEBASE)_log_gen.c"}
    extraFiles+={"$(OUTPUTDIR)/$(OUTPUTNAMEBASE)_log_gen.h"}
    command="Perl Script"
    commandLine="$COMMAND ${scriptdir}/pb_protocol.pl $INPUTFILE \
        $(OUTPUTDIR)/$(OUTPUTNAMEBASE)_gen.c \
        $(OUTPUTDIR)/$(OUTPUTNAMEBASE)_gen.h"

    progress="Processing"
    action("&Process"
    color="#500050"
    findLinkLine=false
}
{
    name="PBCommands"
    outputType="SourceFile"
    grepable=true
    extraFiles={"docs_gen.txt"}
    command="Perl Script"
    commandLine="$COMMAND ../../src/probe/pb_ti_cmds.pl \
        --html='$(OUTPUTDIR)' $INPUTFILE"

    progress="Processing"
    action("&Process"
    color="#500050"
    findLinkLine=false
}
}
```



```
{
    name="PerlScript"
    outputType="SourceFile"
    grepable=true
    command="Perl Script"
    commandLine="$COMMAND $INPUTFILE $OPTIONS"
    progress="Running"
    action("&Run"
    color="#500050"
    findLinkLine=false
}
```

Example 2. Converting Images for Linking

The following example demonstrates how to convert graphical images into a form that can be linked into a project, group the converted images together, and then link them in to the final executable.

There are three file types that can be used to convert images:

- **Resource** — The images.
- **ResourceBundle** — A collection of images and other bundles.
- **NestedBundle** — A collection of images that must be included in a resource bundle.

These files are processed by two commands. Because different options are available for the various bundle types, there are three command definitions.

```
%if ! defined (SDK_BIN_DIR)
% define SDK_BIN_DIR __DIR__
%endif

CommandOptions {
  ResourceConverterOptions {
    ResourceID {
      name="id"
      merge="Replace"
      delimiter="Equal"
      optionType="String"
      guiLabel="Set Resource ID"
      guiCategory="Resource Converter"
      pass={"All"}
      commonLevel=101
    }
    ResourceName {
      name="name"
      merge="Replace"
      delimiter="Equal"
      optionType="String"
      guiLabel="Set the resource"
      guiCategory="Resource Converter"
      pass={"All"}
      commonLevel=101
    }
  }
}
```

```
OutputFormat {
{
  name="default" # option name (stored in .gpj files)
  value=0
  delimiter="NoArg"
  enumLabel="Default type to that in bitmap"
  flags={"FAKEOPTION"} # don't pass on command line
}
{
  name="type=BITMAP_TYPE_16BPP_565"
  value=1
  enumLabel="BITMAP_TYPE_16BPP_565"
}
{
  name="type=BITMAP_TYPE_1BPP_VERTICAL"
  value=2
  enumLabel="BITMAP_TYPE_1BPP_VERTICAL"
}
{
  name="type=BITMAP_TYPE_1BPP_IDEAL"
  value=3
  enumLabel="BITMAP_TYPE_1BPP_IDEAL"
}
  delimiter="NoArg"
  merge="Replace"
  optionType="Enum"
  guiLabel="Set the bitmap output type"
  guiCategory="Resource Converter"
  commonLevel=101
}
}
```

```
NestedBundlerOptions {
  NestedLocation {
    name="-i"
    merge="Replace"
    delimiter="space"
    optionType="String"
    guiLabel="Set Resource ID"
    guiCategory="Resource Bundler"
    pass={"All"}
    commonLevel=101
  }
}

Commands {
  {
    name="Resource Converter"
    exec="${SDK_BIN_DIR}\\imgconvert.exe"
    options={"SpecialOptions", "ResourceConverterOptions"}
  }
  {
    name="resource_bundler"
    exec="${SDK_BIN_DIR}\\resourcebundler.exe"
    options={"SpecialOptions"}
  }
  {
    name="nested_bundler"
    exec="${SDK_BIN_DIR}\\resourcebundler.exe"
    options={"SpecialOptions", "NestedBundlerOptions"}
  }
}
```

```
FileTypes {
    Resource_Bundle {
        name="Resource Bundle"
        outputExtension="rsc"
        command="resource_bundler"
        outputType="Program"
        commandLine="$COMMAND -f $OBJECTS -o $OUTPUTFILE $OPTIONS"
        progress="Bundling"
        action("&Build"
        color="#c00000"

# Nested Bundles can be linked to make Resource Bundles.
    linkableTypes += {"Nested Bundle"}
# Indicates that there's also a header file which is output.
    extraFiles={"$(OUTPUTDIR)\\$(OUTPUTNAMEBASE).h"}
    }
    Nested_Bundle {
        name="Nested Bundle"
        extensions={"gpj"}
        outputExtension="robj"
        command="nested_bundler"
        outputType="Program"
        commandLine="$COMMAND -f $OBJECTS -o $OUTPUTFILE $OPTIONS"
        progress="Bundling"
        action("&Build"
        color="#c00000"

# Nested Bundles can be linked to make Nested Bundles.
    linkableTypes += {"Nested Bundle"}
    }

# These two 'Program.' lines below include the resource bundle
# into an executable program if you have a build structure like
# the following:
#     default.gpj
#         my_program.gpj  [Program]
#         my_resources.gpj [Resource Bundle]
#     my_code.gpj  [Subproject]
# Without these lines, the Resource Bundle will not
# be linked in to the final executable image.
#     Program.linkSpecifiers += {"Resource Bundle; \
#         -extra_file=$(OUTPUTFILE)"}
#     Program.linkableTypes += {"Resource Bundle"}
```

```
{
    name="Resource"
    extensions={"bmp","jpg","jpeg","png","tiff"}
    command="Resource Converter"
    outputExtension="rojb"
    outputFile="ObjectFile"
    commandLine="$COMMAND $INPUTFILE output=$OUTPUTFILE \
        $OPTIONS"
    progress="Converting"
    action("&Convert"
    grepable=false
    color="#006000"
}
```

Customization File Syntax

The following sections provide more details about the syntax of the customization files.

FileType Syntax

Commonly used FileType properties include:

- **action** — A text string displayed in the Project Manager's context menus that represents the process of building this file type. For example, the action for the compiler is **Compile**.
- **color** — The hexadecimal representation of the color to use when displaying files of this type in the Project Manager.
- **command** — The name of the command used to build files of this type. The name of the command must match the name property of a definition in the Commands section.
- **commandLine** — The exact command line used to build files of this type. Numerous variables are available for context-sensitive values. For more information see "Context Sensitive Variables" on page 858.
- **extensions** — A list of filename extensions to associate with this file type (see "File Types" on page 39).

The `extensions` property is optional. If you do not have extensions, you must explicitly mention the `FileType` in the `.gpj` file for each file of that type so that the builder can recognize the `FileType`.

- `extraFiles` — A list of additional output files that should be removed when cleaning output files.
- `findLinkLine` — Specifies whether this file should be included in the command line when linking its parent executable. If this is `true`, directs the Project Manager to look for a parent executable that explicitly accepts this type as a `linkableType`. This property overrides the default behavior based on the `outputType`. The default value of this property is `false`.
- `graphicalEditor` — The name of the command used to edit files of this type. The name must match the name property of a definition in the Commands section. This property is useful if you want to edit the file with a tool other than the MULTI Editor.
- `graphicalEditorCommand` — The exact command line used to edit files of this type. Numerous variables are available for context-sensitive values. For more information see “Context Sensitive Variables” on page 858.
- `grepable` — Specifies whether or not this file can be searched. By default, all files can be searched.
- `linkableTypes` — A list of file type names that are included on the linker command line when linking this file. This property only applies if this file is linked.
- `linkSpecifiers` — A list of file type names and the option to include on the linker command line when linking this file. This property only applies if this file is linked.
- `name` — A string used to identify the file type in the Project Manager. This property is required.
- `outputExtension` — The filename extension on the primary output file generated by files of this type. The Project Manager uses this information to determine if the file must be built. It also uses this information when cleaning output files (**-clean**).
- `outputType` — The class of output file generated by building files of this type. The following list provides all values for `outputType`, and explains what each value indicates to the Project Manager:
 - `ObjectFile` — Building this file produces an object file that is linked into its build parent by default.

- `SourceFile` — Building this file produces a source file that should not be linked into the parent by default.
- `Program` — This file is a container that takes linkable children as input.
- `Library` — This file is an archive that may be incrementally linked. It takes linkable children as input.
- `PassToLinker` — This file is not built, but by default is linked as-is into its build parent.
- `SelectOne` — Only one of this file's children is built, based on the value of **:select** options that are set.
- `Subproject` — This file type is a container, but the file itself is not built. Children may still be linked into this file's build parent.
- `None` — This file is not built and is not linked into its parent.
- `preExec`, `postExec` — Specifies a command line to run before or after processing all files of this type. If you are running a parallel build, this command triggers a choke point.
- `preExecSafe`, `postExecSafe` — Specifies a command line to run before or after processing all files of this type. If you are running a parallel build, this command does not trigger a choke point.
- `progress` — The string prefix shown in the progress messages and **Build Details** dialog box when building files of this type.
- `promoteToFirstPass` — Specifies whether processing this file should occur during the first pass of a multi-pass build. Set this to property to **true** for file types that generate source code that are built into your project.

Context Sensitive Variables

When defining some of the properties in the `FileTypes` section, you might need to specify information that depends on the specific file being processed. Because the file type definition is generic, you can use the following variables as placeholders for the context-specific information.

The syntax for using these variables is:

- `$VARIABLE`
- `$(VARIABLE)` — This format is required if the variable name contains an underscore.

To specify a dollar sign in a property value, use `$$`. The supported variables are:

COMMAND	The executable from the <code>Commands</code> definition that matches the <code>command</code> property of the <code>FileType</code> . Resolves to an absolute path if the <code>runInDirectory</code> flag is set.
DISKFILE	Full path to the file being processed.
EDITOR	Executable of the graphical editor defined by the <code>FileType</code> of the file being processed.
FILENAME	The name of the file being processed. Does not include path information.
FILENAMEBASE	Base name with no suffix of the file being processed.
FILENAMESLIBBASE	The base name of a library. Same as <code>FILENAMEBASE</code> except that it strips away the <code>lib</code> prefix if present.
FILETYPEOPTIONS	List of file types included in the current module. This is used by the driver to determine when a module requires C or C++ libraries.
INPUTFILE	The name of the input file being processed, including path information relative to the Top Project directory. For commands with multiple passes, this variable also includes a list of output files from the first pass when used with the second pass.
INPUTDIR	Relative path to the directory containing the input file.
ITEMID	Unique ID number for the file being processed. This number is not guaranteed to be the same across builds.
KEEPOBJECTS	List of all objects in an archive that should remain in the archive after processing.
MAKE_DEPEND_OPTIONS	Arguments specified in the <code>makeDepends</code> section of the <code>COMMAND</code> definition used for generating dependency information for the file being processed.
OBJECTS	List of all the objects to be linked together for this file.

OBJECTBASE
Base name with no suffix of the last object file in the list. This is a special case for single file libraries.
OBJECTSUFFIX
Suffix for the object files.
OPTIONS
A space-separated string containing all of the options set on this file, including inherited options.
OUTPUTDIR
Relative path to the output directory.
OUTPUTFILE
Relative path to the primary file output by processing the current file (that is, the object file for a source file). If the file requires two-pass processing, the extension on the OUTPUTFILE is different for the two passes.
OUTPUTNAMEBASE
Base name with no suffix of the output name for the file being processed.
PARENTID
Unique ID number for the parent of the file being processed. If there is no parent, the value is zero. This number is not guaranteed to be the same across builds.
RUNDIR
The directory the command is run in. This is the same as the directory containing the Top Project except for commands with the <code>runInDirectory</code> flag set.
SECOND_PASS_OPTION
Same as the <code>secondPassOption</code> from the <code>FileType</code> definition. Used for tools that operate using multiple passes to indicate the current pass.
TOPPROJECT
Full path of the Top Project.
SIBLINGS
Full paths of all files of the <code>siblingType</code> , which are descendents of the parent of the file being processed.
SIBLINGS <i>[type]</i>
Like SIBLINGS, but uses the specified type instead of the <code>siblingType</code> parameter.

Command Syntax

Commonly used Command syntax includes:

- `exec` — The name of the program to run. If it contains spaces, the Builder treats the entire string as the program name (it does not parse the string into arguments). If the program name is specified without a directory, the builder searches for the program in the directories of the `PATH` environment variable. Required.
- `name` — A string used to identify the command in `FileType` definitions. Required.
- `options` — A list of option groups supported by this command. All commands should support `SpecialOptions`. Required.
- `specialOptions` — Flags supported by the program that can be used for integration with advanced features in the Project Manager. Two commonly used special properties are:
 - `runInDirectory` — Specifies whether the command should be run with the current working directory set to the directory containing the input file. By default, the command is run with the working directory set to the directory containing the Top Project.
 - `showCommands` — Specifies the flag used to generate verbose output.

CommandOption Syntax

The following values are available for `CommandOption` property definitions:

- `commonLevel` — Specifies how visible the option is:
 - `1 - 100` — A common option, which appears in the **Basic Options** tab as well as the **All Options** tab.
 - `101 - 1000` — A standard option, which appears on the **All Options** tab.
 - `>1000` — [default] An advanced option, which appears on the **All Options** tab in the **Advanced** option group.
- `delimiter` — Specifies the connector between the option and any parameter it may take:
 - `Equal` — Is in the form *-option=value*.

- NoArg — [default] Is in the form *-option*.
- Space — Is in the form *-option value*.
- Touching — Is in the form *-optionvalue*.
- merge — Specifies how multiple settings of the option are defined in the same or different files:
 - Concat — Appends a value to the end of a list.
 - None — [default] The value is not inherited.
 - Preconcat — Prepends a value to the beginning of a list.
 - Replace — Replaces the previous value with the new one.
- name — The exact string or prefix to pass on the command line. This string must not contain spaces. Required.
- optionType — Specifies the type of option:
 - Enum — An on/off switch, and permits a merge type of Replace.
 - EnumList — A group of on/off switches, and permits a merge type of Concat.
 - List — [default] Accepts a comma-separated list of values and permits a merge type of Concat or Preconcat.
 - String — Accepts a single value and permits a merge type of Replace.
- pass — Specifies which parts of the toolchain the option is passed to:
 - All — [default] All tools (most common).
 - Compile — Compiler only.
 - Depends — Preprocessor dependencies resolution only.
 - Link — Linker only.
 - None — None of the tools.
- value — A numerical identifier for this option setting. If you have two sub-definitions with different spellings but the same meaning, specify the same numerical identifier for their `value` fields. If they have different meanings, use different numerical identifiers for each `value` field.

Preprocessor Directives

Like C and C++ source files, the customization files are processed by a preprocessor before they are loaded by the Builder. The preprocessor supports numerous standard directives you can use to conditionally include or disable portions of your customization file:

<code>%define</code> Defines a macro.
<code>%echo</code> <code>%info</code> Outputs a message.
<code>%elif</code> <code>%elsif</code> <code>%elseif</code> <code>%else</code> <code>%endif</code> <code>%if</code> Conditional inclusion directives. <code>%elif</code> and <code>%elsif</code> are equivalent to <code>%elseif</code> .
<code>%error</code> Outputs an error message.
<code>%ifdef</code> Shorthand for <code>%if defined</code> .
<code>%include</code> Includes another file.
<code>%warning</code> Outputs a warning message.

Preprocessor Functions

In addition to the preprocessor directives listed in the preceding section, you can use several preprocessor functions in your customization file. These functions are typically used in combination with the `%if` directive.

- `defined(MACRO_NAME)` — Evaluates to true if the macro is defined.
For example:

```
%if defined(FOO)
```

- `file_exists(FULL_PATH)` — Evaluates to true if the file or directory specified exists. For example:

```
%if file_exists("${__DIR__}/myfile")
```

- `streq(STRING1,STRING2)` — Evaluates to true if the strings are equal. For example:

```
%if streq("${__MULTI_HOST__}", "linux86")
```

Predefined Macros

You can use the following macros in any customization file:

- `__DIR__` — Directory of the file you are editing.
- `__INTEGRITY_DIST__` — Directory of the INTEGRITY installation (only defined for INTEGRITY targets).
- `__MULTI_HOST__` — The host machine's operating system.
- `__MULTI_DIR__` — Directory of the MULTI installation.

The syntax for using one of these macros is `${MACRO}`. See Example 1 on page 864 for an example.

Example 1. Including Additional .bod files

The following example demonstrates how to use preprocessor directives, functions, and macros to include additional, optional **.bod** files.

```
# Include kernel build types configuration file
%if file_exists("${__DIR__}/../kernel/kernel.bod")
#include "${__DIR__}/../kernel/kernel.bod"
%endif

# Include build type configuration file for examples
#include "${__DIR__}/../examples/copyfile.bod"

# Include build type configuration file for validations
%if file_exists("${__DIR__}/../rtos_val/validations.bod")
#include "${__DIR__}/../rtos_val/validations.bod"
%endif
```

Third-Party License and Copyright Information

Appendix D

ldpreload.c Copyright C 2001 Steven Engelhardt All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

- * (asterisk) wildcard character, 33
- * (pointer) data type, 106
 - signedness, 226
- ? operator, 626
- ? wildcard character, 33
- # driver option, 95, 830
- ### legacy GNU option, 830
- 64-bit integer arguments, 710

A

- a driver option, 119, 188
- A driver option, 254
- .a file extension, 41, 88, 474
- abort(), ANSI C implementation, 618
- About MULTI Project Manager** Project Manager menu item, 80
- ABS linker section attribute, 451
- __abs() intrinsic function, 695
- absolute assembler expression, 406
- absolute linker expression function, 449
- Accept GNU __asm__ statements** Builder option, 305
- acos(), ANSI C implementation, 614
- acosf(), ANSI C implementation, 614
- acosh(), ANSI C implementation, 614
- Add Extra File to Debug Info** Builder option, 296
- Add File after selected file** Project Manager menu item, 68
- Add File After selected file** Project Manager menu item, 70

- Add File into selected file** Project Manager menu item, 68
- Add Item into selected file** Project Manager menu item, 68
- adding executables and examples Project Manager, 26
- Additional Assembler Options** Builder option, 242, 392
- Additional Intex Options** Builder option, 170
- Additional Linker Options (among object files)** Builder option, 249
- Additional Linker Options (before start file)** Builder option, 248, 433
- Additional Linker Options (beginning of link line)** Builder option, 248
- Additional MakeRPL Arguments** Builder option, 165
- Additional Output Files** Builder option, 284
- Additional PrepRPL Arguments** Builder option, 165
- Additional SDA Base Registers** Builder option, 145, 269
- Additional System Library** Builder option, 162
- additional_sda_reg driver option, 145, 269
- addr linker expression function, 449
- Advanced Build** Project Manager menu item, 74
- Advanced** Project Manager menu item, 70
- .ael output file type, 93

Index

- alias analysis
 - interprocedural optimizations, 330
 - library function, 331
 - read-based, 330
 - write-based, 330
- .align assembler directive, 415
- align linker expression function, 449
- ALIGN linker section attribute, 451
- alignment
 - bitfields, 594
 - packing, *see* structure packing
 - requirements, 106, 777
 - type double, 674
- alignof linker expression function, 449
- allabsolute linker option, 434
- alloca() function, 711
- Allocation of Uninitialized Global Variables** Builder option, 227, 682, 684
- Allow C++ Style Slash Comments in C** Builder option, 196
- Allow Use of SPE instructions** Builder option, 271
- allow_bad_relocations linker option, 434
- allow_different_section_types driver option, 311
- Alternate Assembler Directory** Builder option, 287
- Alternate Compiler Directory** Builder option, 286
- Alternate Compression Table** Builder option, 268
- Alternate Header Name** Builder option, 314
- Alternate Library Directory** Builder option, 286
- Alternate Linker Directory** Builder option, 287
- alternate program start address
 - specifying, 147
- Alternate Source Name** Builder option, 268
- Alternative Tokens** Builder option, 228
- alternative_tokens driver option, 229
- Altivec driver option, 830
- __ALTIVEC__, 668
- Always Use lmw/stmw in Interrupt Subroutine Prologue/Epilogue** Builder option, 272
- Annotated C++ Reference Manual, The (ARM)*, 630
- annotated examples
 - images for linking, converting, 851
- ANSI Aliasing Rules** Builder option, 196
- ANSI C
 - extensions, 565
 - implementation-defined features
 - abort(), 618
 - acos(), 614
 - acosf(), 614
 - acosh(), 614
 - arrays, 610
 - asin(), 614
 - asinf(), 614
 - asinh(), 614
 - assert(), 614
 - atan2(), 614
 - atan2f(), 614
 - atanh(), 614
 - bit-fields, 611
 - calloc(), 618
 - character set, execution, 615
 - characters, 608
 - declarator, 613
 - environment, 607
 - errno(), 618
 - exit(), 618
 - files, 616
 - floating-point, 610
 - fmod(), 615
 - getenv(), 618

Index

- identifiers, 607
- #include directive, 613
- integers, 609
- isalnum(), 614
- isalpha(), 614
- iscntrl(), 614
- islower(), 614
- isprint(), 614
- isupper(), 614
- locale, 615
- log(), 615
- logf(), 615
- malloc(), 618
- NULL, 614
- overview, 605
- pointers, 610
- pow(), 615
- powf(), 615
- registers, 611
- signals, 616
- sqrt(), 615
- sqrtf(), 615
- sterror(), 619
- streams, 616
- strftime(), 615
- structures, 611
- switch statements, 613
- tmpfile(), 617
- translation, 606
- underflow detection, 615
- unions, 611
- volatile accesses, 613
- limitations, 592
- required macro names, 664
- specifying, 565
- ansi** driver option, 194, 564 to 565, 587
- ANSI** driver option, 194, 564, 572, 588
- ansi** header file directory, 705
- ansi_alias** driver option, 197
- any_output_suffix** driver option, 275
- Append Comment Section with**
 - Link-Time Information** Builder option, 246
 - Append Default Extension to Output Filename** Builder option, 278
 - append** linker option, 434
 - Append String to Every Section Name** Builder option, 304
 - :appendExtension** Builder option, 278
 - archive** driver option, 91 to 92, 474
 - archiver, *see* librarian
 - archives, *see* libraries
 - argument field, 399
 - arithmetic
 - machine specific, 782
 - pointer, 780
 - ARM compliant C++, 630
 - arm** driver option, 195, 630
 - __ARRAY_OPERATORS**, 675
 - arrays
 - alignment of, 105
 - new and delete, predefined macro, 675
 - .ascii** assembler directive, 417
 - .asciz** assembler directive, 417
 - asin()**, ANSI C implementation, 614
 - asinf()**, ANSI C implementation, 614
 - asinh()**, ANSI C implementation, 614
 - asm** driver option, 242, 392
 - asm facility, 788
 - asm body, 794
 - asm macro, 789
 - con, 793
 - error, 793
 - examples, 789
 - farmem, 794
 - farsprel, 794
 - lab, 794
 - macro body, 789
 - mem, 794
 - nearmem, 794
 - pattern, 789

Index

- reg, 794
- storage mode, 789
- treg, 793 to 794
- ureg, 794
- using asm macros, 791
- writing asm macros, 798
- .asm** file extension, 40, 88
- asm keyword, *see* asm statements
- __asm keyword, *see* asm statements
- asm statements
 - overview, 598
 - #pragma directive for, 680
- Asm Statements** Builder option, 262, 588, 599
- asm_errors** driver option, 262, 599
- asm_silent** driver option, 262, 588, 599
- asm_warnings** driver option, 262, 599
- asm3g** driver option, 243
- asmcmd** driver option, 243, 392
- asppc**, *see* assembler
- assembler, 392
 - assignment statements, 401
 - command line options for, 394
 - escape sequences, 398
 - expressions, 401
 - introduction to, 8
 - invoking with the driver, 125
 - invoking with the Project Manager, 125
 - list file output, 409
 - listings, 424
 - passing options to through compiler driver, 392
 - running with compiler driver, 392
 - syntax, 396
- Assembler Command File** Builder
 - option, 243, 392
- assembler directives
 - .align, 415
 - .ascii, 417
 - .asciz, 417
 - .bss, 421
 - .byte, 417
 - .comm, 426, 774
 - .data, 421
 - .double, 417
 - .dsect, 426, 771
 - .eject, 424
 - .else, 416
 - .elseif, 416
 - .end, 426, 771
 - .endif, 416
 - .endm, 420
 - .endr, 421
 - .equ, 401, 426
 - .error, 428
 - .exitm, 420
 - .extern, 426, 770
 - .file, 427
 - .float, 417
 - .fsize, 427
 - .gen, 424
 - .ghsnote, 471
 - .global, 426
 - .globl, 426, 770
 - .ident, 428
 - .if, 416
 - .include, 418
 - .inspect, 426
 - .lcomm, 426
 - .list, 424
 - .long, 417
 - .lsbss, 427
 - .macro, 420
 - .need, 429
 - .nogen, 424
 - .nolist, 425
 - .note, 421
 - .novle, 429
 - .nowarning, 429
 - .offset, 417
 - .org, 421
 - .previous, 421

Index

- .rept, 421
- .rodata, 422
- .sbss, 427, 774
- .sbttl, 425
- .scall, 428
- .sdata, 422
- .sdata2, 422
- .section, 423
- .set, 401, 427
- .short, 418
- .size, 428
- .skip, 418
- .space, 418
- .strz, 418
- .subtitle, 425
- .text, 423
- .title, 425
- .type, 428
- .vle, 429
- .warning, 429
- .weak, 427
- assembler options
 - b, 394
 - b0, 394
 - b1, 394
 - cpu=*cpu*, 394
 - help, 394
 - I, 394
 - list, 395
 - nogen, 395
 - noSPE, 394
 - o, 395
 - r, 395
 - ref, 395
 - regs, 394
 - SPE, 394
 - V, 395
 - w, 395
- Assembler Warnings Builder option, 309
- assembler_warnings driver option, 309
- assembly language, 396
 - assignment statements, 401
 - comments, 399
 - constants for, 397
 - continuation lines, 400
 - escape sequences for, 398
 - expression types, 406
 - expressions, 401
 - identifiers for, 396
 - interface, 781
 - interleaving with source code, 242
 - labels, 407
 - line terminators, 400
 - operator precedence, 402
 - operators, 401
 - source statement syntax, 399
 - type combinations, 406
- assert(), ANSI C implementation, 614
- assignment statements, assembler, 401
- AT linker section attribute, 451
- atan2(), ANSI C implementation, 614
- atan2f(), ANSI C implementation, 614
- atanh(), ANSI C implementation, 614
- auto_instantiation driver option, 239
- Auto-Instantiation of Templates** Builder option, 239
- auto_sda driver option, 141, 146, 158
- :autoInclude driver option, 285
- Automatic Link-Time SDA** Builder option, 141, 146, 157, 178
- Automatic Register Allocation** Builder option, 289, 363
- automatic register allocation
 - optimization, 363
- automatic two-pass inlining, 321
- Automatic Vector Optimization** Builder option, 178, 344
- automatically including files
 - Project Manager, 41
- autoregister driver option, 289
- AutoSDA, *see* -auto_sda driver option
- ax, *see* librarian

Index

B

- b assembler option, 394
- b0 assembler option, 394
- b1 assembler option, 394
- __BASE__, 674
- __BIG_ENDIAN__, 669
- bigendian driver option, 154, 830
- __bigendian keyword, 589
- bigswitch driver option, 273
- binary code generation, 8, 274
- Binary Code Generation** Builder option, 273
- binary memory image
 - generating, 244, 524
- binary operators, 401
- Binary Output Directory Relative to This File** Builder option, 276
- Binary Output Directory Relative to Top-Level Project** Builder option, 276
- :binDir Builder option, 276
- :binDirRelative Builder option, 276
- bitfields, 592, 778
 - alignment and size, 594
 - code efficiency, 593
 - portability, 778
 - signedness of, 225, 592
- bl instruction, 107
- block count profiling, *see* coverage profiling
- blr instruction, 107
- .bmon output file type, 93
- .bod files, *see* customization files (.bod)
- Bookmarks** Project Manager menu item, 80
- __BOOL, 675
- bool data type, 675
- bool driver option, 230, 675
- Bool Type Support** Builder option, 230
- Brief Error Message Mode** Builder option, 260
- brief_diagnostics driver option, 260

- .bsp
 - file extension, 41
- bsp driver option, 113
- .bss assembler directive, 421
- .bss program section, 127, 813
- buffering
 - C++ I/O streams, 710
- Build Ignoring Errors** *selected file* Project Manager menu item, 73
- build mode, setting, 47
- Build** *selected file* Project Manager menu item, 73
- Builder options
 - Accept GNU __asm__ statements**, 305
 - Add Extra File to Debug Info**, 296
 - Additional Assembler Options**, 242, 392
 - Additional Intex Options**, 170
 - Additional Linker Options (among object files)**, 249
 - Additional Linker Options (before start file)**, 248, 433
 - Additional Linker Options (beginning of link line)**, 248
 - Additional MakeRPL Arguments**, 165
 - Additional Output Files**, 284
 - Additional PrepRPL Arguments**, 165
 - Additional SDA Base Registers**, 145, 269
 - Additional System Library**, 162
 - Allocation of Uninitialized Global Variables**, 227, 682, 684
 - Allow C++ Style Slash Comments in C**, 196
 - Allow Use of SPE instructions**, 271
 - Alternate Assembler Directory**, 287
 - Alternate Compiler Directory**, 286
 - Alternate Compression Table**, 268
 - Alternate Header Name**, 314

Index

- Alternate Library Directory**, 286
- Alternate Linker Directory**, 287
- Alternate Source Name**, 268
- Alternative Tokens**, 228
- Always Use Imw/stmw in Interrupt Subroutine Prologue/Epilogue**, 272
- ANSI Aliasing Rules**, 196
- Append Comment Section with Link-Time Information**, 246
- Append Default Extension to Output Filename**, 278
- Append String to Every Section Name**, 304
- :appendExtension**, 278
- Asm Statements**, 262, 588, 599
- Assembler Command File**, 243, 392
- Assembler Warnings**, 309
- Auto-Instantiation of Templates**, 239
- Automatic Link-Time SDA**, 141, 146, 157, 178
- Automatic Register Allocation**, 289, 363
- Automatic Vector Optimization**, 178, 344
- Binary Code Generation**, 273
- Binary Output Directory Relative to This File**, 276
- Binary Output Directory Relative to Top-Level Project**, 276
- :binDir**, 276
- :binDirRelative**, 276
- Bool Type Support**, 230
- Brief Error Message Mode**, 260
- C Include Directories**, 302
- C Japanese Automotive Extensions**, 195, 587 to 588
- C Language Dialect**, 194, 564 to 565, 572 to 573, 579, 582, 587 to 588, 590, 592, 600
- C++ Exception Handling**, 196, 623 to 624
- C++ Include Directories**, 124, 302
- C++ Inlining Level**, 303, 328
- C++ Language Dialect**, 195 to 196, 622 to 624, 627, 629 to 631
- C++ Libraries**, 196, 623
- C++ Linking Method**, 308
- C++ Standard Include Directories**, 302
- C/C++ Minimum/Maximum Optimization**, 290, 359
- Call Graph Generation**, 257
- Checksum**, 150, 258
- Code Factoring**, 251, 469
- Commands to Execute After Associated Command**, 281
- Commands to Execute After Associated Command (via Shell)**, 282
- Commands to Execute Before Associated Command**, 279
- Commands to Execute Before Associated Command (via Shell)**, 280
- Common Subexpression Elimination**, 289, 355
- Consistent code without debug information**, 293
- Constant Propagation**, 290, 358
- Convert DWARF Files to MULTI (.dbo) Format**, 298
- Convert Stabs Files to MULTI (.dbo) Format**, 299
- Create Raw Binary Image**, 169
- Create RPL/RPX**, 163
- Cross-Reference Information**, 297
- Dbo File Version**, 314
- Debug Information (.dbo) Tracing Diagnostics**, 313
- Debugging Information**, 298

Index

- Debugging Level**, 115 to 116, 187, 288, 298, 690
- default, 50
- Deferral of Function Template Parsing**, 239, 304
- Define Linker Constant**, 253
- Define Preprocessor Symbol**, 192
- Definition of Standard Symbols**, 300
- Definition of Unsafe Symbols**, 300, 664
- Delay Archives**, 292
- Delay Second Pass**, 292
- Delete Unused Kernel Calls**, 169
- Deletion of Unused C++ Virtual Functions**, 250
- Deletion of Unused Data**, 250
- Deletion of Unused Functions**, 126, 250
- Demangling of Names in Linker Messages**, 308
- Dependencies Relative to Source Directory**, 277
- Dependencies Relative to This File**, 277
- Dependencies Relative to Top-Level Project**, 277
- Dependent Name Processing**, 238 to 239, 304, 624, 641, 645
- :depends**, 277
- :dependsNonRelative**, 277
- :dependsRelative**, 277
- Display BootTable**, 169
- Display Error Message Numbers**, 264 to 265, 686
- Display Error Message Paths**, 261
- Display includes Preprocessor Directives Listing**, 192
- Display Local Symbols in Map File**, 256
- Display Unused Functions in Map File**, 256
- Display Version Information**, 260
- Distinct C and C++ Functions**, 308, 624
- DoubleCheck Config File**, 267
- DoubleCheck Errors to Ignore**, 267
- DoubleCheck Level**, 266
- DoubleCheck Output that Stops Builds**, 267
- DoubleCheck Report File**, 266
- Dynamic Download Project**, 168
- EDG Front End Options**, 313
- Emulated GNU Version**, 303
- End Files**, 252
- Endianness**, 154
- Event Logging**, 166
- Executable Stripping**, 245
- Expansion of e500 evmwhgs__a__ Intrinsics**, 270
- Expansion of e500 evmwhs__a_w Intrinsics**, 270
- Extend Variable Liveness**, 294
- :extraOutputFile**, 284
- Far Calls**, 136, 158
- far function calls, 136
- Files Listing External Symbols**, 181
- Files to Pre-Include**, 300
- Floating-Point Coprocessor**, 155
- Floating-Point Precise Signed Zero**, 156
- Floating-Point Truncation**, 156
- Follow Section**, 312
- Force All Symbols From Library**, 254
- Force Deletion of Unused C++ Virtual Function Symbol**, 311
- Force Frame Pointer**, 295
- Force Undefined Symbol**, 254
- Full Breakdots**, 294
- Full POSIX-2001 Compilation Support**, 168
- Function Prologues**, 272, 685

Index

- Functions Without Prototypes**, 261, 263
- Generate Additional Output**, 92, 244
- Generate MULTI and Native Information**, 298
- Generate Target-Walkable Stack**, 293
- Generated Header File Directory**, 170
- Generated Header File Name**, 170
- Generation of fsel Instructions**, 270
- Generation of isel Instructions**, 270
- Global Registers**, 134, 146, 160
- GNU Compatibility Optimizations**, 291
- Guiding Declarations of Template Functions**, 235
- Host & Target Character Encoding**, 229, 603, 657
- HTML File Compression**, 268
- Identifier Definition**, 309
- Identifier Support**, 309
- Implicit Int Return Type**, 263
- Implicit Source File Inclusion**, 236, 239, 624, 640 to 641
- Implicit Use of 'std' Namespace**, 230
- Import Symbols**, 254
- Include Directories**, 61, 122 to 124, 172
- Include File Search Directory**, 170
- Include libbsp.a**, 167
- Incorrect Pragma Directives**, 262, 679
- inheriting**, 62
- Inline C Memory Functions**, 288, 326
- Inline C String Functions**, 288, 326
- Inline Larger Functions**, 180, 324
- Inline Specific Functions**, 186, 322 to 324
- Inline Tiny Functions**, 287, 317
- Input Languages**, 88, 275
- Instantiate Extern Inline**, 231, 624
- INTEGRITY Application Stripping**, 171
- INTEGRITY Posix Mode**, 168
- INTEGRITY Relocatable Program**, 167
- Interleaved Source and Assembly**, 242
- Intermediate Output Directory Relative to This File**, 276
- Intermediate Output Directory Relative to Top-Level Project**, 175
- Intermodule Inlining**, 178, 316, 321, 323 to 324
- Interprocedural Optimizations**, 179, 330, 334, 336 to 338
- Intex Variable Definition**, 171
- IP Alias Lib Functions**, 185, 331
- IP Alias Reads**, 184, 330
- IP Alias Writes**, 184, 331
- IP Constant Globals**, 182, 336
- IP Constant Propagation**, 183, 335
- IP Constant Returns**, 184, 331
- IP Delete Functions**, 182, 336
- IP Delete Globals**, 182, 336
- IP Limit Inlining**, 183
- IP One-Site Inlining**, 182, 335
- IP Remove Parameters**, 183, 335
- IP Remove Returns**, 184, 335
- IP Small Inlining**, 183, 331
- Kernel Project**, 166
- Libraries**, 125, 173
- Libraries and PrepRPL**, 163 to 164
- Library Directories**, 125, 173
- Line Wrap Messages**, 261
- Link in Standard Libraries**, 310
- Link Mode**, 311
- Link-Once Template Instantiation**, 235, 635, 638
- Link-Time Constant Propagation**, 251

Index

- Link-Time Ignore Debug**
 - References**, 252
- Link-Time Trivial Function**
 - Inlining**, 251
- Linker Command File**, 249
- Linker Directive Files with**
 - Non-standard Extensions**, 247
- Linker Directives Directory**, 310
- Linker Optimizations**, 178, 250
- Linker Warnings**, 246
- Linker-Based Far Call Patching**, 135, 159
- Linking Sections with Different**
 - Types**, 311
- Long Long Support**, 226
- longjmp() Does Not Restore Local**
 - Vars**, 304
- Loop Optimize Specific**
 - Functions**, 186, 340
- Loop Unrolling**, 288, 339, 342
- Map File Cross-Referencing**, 256
- Map File Generation**, 255
- Map File Page Length**, 255
- Map File Retention**, 255
- Map File Sorting**, 256
- Maximize Optimizations**, 180
- Maximum Number of Errors to**
 - Display**, 259
- Memory Optimization**, 290, 359 to 360, 590, 784 to 785
- Message Pragma**, 301
- MISRA C - Advisory Rules**
 - Level**, 215
- MISRA C - Required Rules**
 - Level**, 215
- MISRA C - Run-Time Checks**, 216
- MISRA C 1998 Rules - Character**
 - Set**, 217
- MISRA C 1998 Rules -**
 - Comments**, 217
 - Constants**, 218
 - Control Flow**, 221
 - Conversions**, 220
 - Declarations & Definitions**, 218
 - Environment**, 216
 - Expressions**, 220
 - Functions**, 221
 - Identifiers**, 217
 - Initialization**, 219
 - Operators**, 219
 - Pointers and Arrays**, 223
 - Preprocessor**, 222
 - Standard Library**, 224
 - Structs and Unions**, 223
 - Types**, 218
- MISRA C 2004 Rules - 1**
 - Environment**, 200
- MISRA C 2004 Rules - 10 Arithmetic**
 - Type Conversions**, 205
- MISRA C 2004 Rules - 11 Pointer**
 - Type Conversions**, 206
- MISRA C 2004 Rules - 12**
 - Expressions**, 206
- MISRA C 2004 Rules - 13 Control**
 - Statement Expressions**, 208
- MISRA C 2004 Rules - 14 Control**
 - Flow**, 209

Index

- MISRA C 2004 Rules - 15 Switch Statements**, 210
- MISRA C 2004 Rules - 16 Functions**, 210
- MISRA C 2004 Rules - 17 Pointers and Arrays**, 211
- MISRA C 2004 Rules - 18 Structs and Unions**, 212
- MISRA C 2004 Rules - 19 Preprocessing Directives**, 212
- MISRA C 2004 Rules - 2 Language Extensions**, 200
- MISRA C 2004 Rules - 20 Standard Libraries**, 214
- MISRA C 2004 Rules - 21 Run-time Failures**, 215
- MISRA C 2004 Rules - 3 Documentation**, 201
- MISRA C 2004 Rules - 4 Character Sets**, 201
- MISRA C 2004 Rules - 5 Identifiers**, 202
- MISRA C 2004 Rules - 6 Types**, 203
- MISRA C 2004 Rules - 7 Constants**, 203
- MISRA C 2004 Rules - 8 Declarations and Definitions**, 203
- MISRA C 2004 Rules - 9 Initialization**, 205
- MULTI Version**, 294
- Multiply-Defined Symbols**, 253
- Namespace Support**, 230, 645
- NaN Compares as Unordered**, 156
- New-Style Cast Support**, 232
- :nodepends**, 278
- Non-Standard Output Suffix**, 275
- :noSelfDepend**, 277
- Object File Output Directory**, 172, 237
- Olimit= Without Debug Information**, 293
- One Instantiation Per Object**, 236 to 237, 637
- Only explicit floating point and SIMD register use**, 272
- Optimization Strategy**, 50, 142, 176, 181, 185, 231, 272 to 273, 287, 316, 326, 329, 351 to 352, 355 to 356, 358 to 362, 590, 690, 785
- Optimize All Appropriate Loops**, 291, 339, 344, 785
- Optimize Specific Functions for Size**, 185
- Optimize Specific Functions for Speed**, 185
- Options to Apply to Project**, 286
- OS Directory**, 163
- Output File Size Analysis**, 257
- Output File Type**, 244
- Output Filename**, 147, 174
- Output Filename for Generic Types**, 278
- :outputDir**, 175
- :outputDirRelative**, 276
- :outputName**, 278
- Package Analysis Files**, 292
- Packing (Maximum Structure Alignment)**, 110 to 111, 226, 689
- Parse Templates in Generic Form**, 238 to 239
- Pass Through Arguments**, 284
- :passThrough**, 284
- Pattern of Files to Pull Into Project**, 285
- Peephole Optimization**, 291, 351
- Physical Address Offset From Virtual Address**, 311
- Pipeline & Peephole Scope**, 180, 351 to 352
- Pipeline Instruction Scheduling**, 291, 352

Index

- Place 'EIEIO' Around Volatile Loads, 161
- Place 'EIEIO' Around Volatile Stores, 161
- Placement of Class Constructor Call to New, 233
- Placement of Zero-Initialized Data, 159
- POSIX DLL, 167
- :postexec, 282
- :postexecSafe, 282
- :postexecShell, 283
- :postexecShellSafe, 283
- :preexec, 279
- :preexecSafe, 280
- :preexecShell, 280
- :preexecShellSafe, 281
- Prelink File to Create Template Instances, 239
- Prelink with Instantiations, 237
- Prepend String to Every Section Name, 304
- Preprocess Assembly Files, 241, 392
- Preprocess Linker Directives Files, 246
- Preprocess Special Assembly Files, 242
- PrepRPL Export Object File, 164
- printf & scanf Argument Type Checking, 263
- Process #include Directives, 299
- Profiling - Call Count, 117, 187
- Profiling - Coverage, 119, 188
- Profiling - Target-Based Timing, 121, 188
- Quit Building if Intex Warnings are Generated, 171
- Quit Building if Warnings are Generated, 260
- Raw Import Files, 246

- Recognition of Exported Templates, 238, 624, 641
- Record Pathname (Not linked), 255
- Redirect Error Output to File, 260
- Redirect Error Output to Overwriting File, 312
- Reference, 285
- Register Allocation by Coloring, 289, 362
- Register Description File, 154
- Relocatable Module, 167
- Relocatable Module Base Image, 168
- Remarks, 259
- Rename Section, 160
- restrict Keyword Support, 231
- Retain Comments During Preprocessing, 301
- Retain Symbols for Unused Statics, 228
- Run-Time Error Checks, 188, 684
- Run-Time Memory Checks, 191
- Run-Time Type Information Support, 233
- Safe Commands to Execute After Associated Command, 282
- Safe Commands to Execute After Associated Command (via Shell), 283
- Safe Commands to Execute Before Associated Command, 280
- Safe Commands to Execute Before Associated Command (via Shell), 280
- Scan source files to augment native debug info, 296
- Search for DBA, 295
- Search Path for .dbo Files, 295
- Section Overlap Checking, 257
- :select, 48, 284
- 'Select One' Project Extension List, 284

Index

- Self-Dependency, 277
- Set Maximum DBO Not Found Warnings, 296
- Set Message to Error, 264
- Set Message to Remark, 265
- Set Message to Silent, 265
- Set Message to Warning, 264
- SHA-1, 171
- Shadow Declarations, 263
- Shared Libraries, 162
- Signedness of Bitfields, 225, 587, 592, 659
- Signedness of Char Type, 225, 587, 609, 658
- Signedness of Pointers, 225
- Simplify C Print Functions, 288
- Source Directories Relative to This File, 174
- Source Directories Relative to Top-Level Project, 276
- Source Directory Is Include Directory, 302
- Source Listing Generation, 125, 241 to 242
- Source Listing Generation Output Directory, 241
- Source Root, 163, 172
- :sourceDir, 174
- :sourceDirNonRelative, 276
- Special Data Area, 138, 140, 158, 688
- Stack Limit Checking, 160
- Standard Include Directories, 301
- Start Address Symbol, 147, 245, 437
- Start File Directory, 252
- Start Files, 252
- Support `__noinline` Keyword, 231, 329
- Support floating point types in `scanf`, 310
- Support for C Type Information in Assembly, 243
- Support for
 - Constructors/Destructors, 232
- Support for Implicit Extern C Type Conversion, 308
- Support for Implicit Typenames, 236
- Support for Old-Style Specializations, 235 to 236
- Support for Very Large Switch Statements, 273
- Suppress Dependencies, 278
- Symbols Referenced Externally, 181
- System Include Directories, 302
- Tail Calls, 290, 356
- Target Processor, 269
- Template Instantiation Output Directory, 237
- Temporary Output, 275
- Temporary Output Directory, 274
- Treat Doubles as Singles, 155
- Treatment of RTTI as `const`, 233
- Treatment of Virtual Tables as `'const'`, 234
- Trigraphs, 263
- types, 49
- Undefine Preprocessor Symbol, 192
- Undefined Preprocessor Symbols, 264
- Undefined Symbols, 253
- Uniquely Allocate All Strings, 228, 657
- Unknown Pragma Directives, 262, 679
- Unroll Larger Loops, 51, 180, 342
- Use Floating-Point in `stdio` Routines, 157
- Use Smallest Type Possible for Enum, 107, 226, 588, 595, 658
- Vector Peeling, 292, 350
- Virtual Function Definition, 234
- VLE Code Generation and Libraries, 162

Index

Wait for Debug Translation Before

Exiting, 297

Warnings, 259, 607

X-Switches, 313

building, *see* Project Manager

built-in functions, *see* intrinsics

`__builtin_return_address()`

intrinsic function, 692

`__builtin_set_return_address()`

intrinsic function, 693

byte addressable, 775, 780

.byte assembler directive, 417

byte order, *see* endianness

`__bytereverse` keyword, 588

C

C, *see* C language

-c driver option, 90, 92, 238

-C driver option, 253, 301

.c file extension, 40, 87

.C file extension, 40, 87

-c HTML compiler option, 268

C Include Directories Builder option, 302

C Japanese Automotive Extensions

Builder option, 195, 587 to 588

C language

and memory-mapped I/O, 149

ANSI dialect, 592

arrays, 771

asm statements, 598

`__bigendian` keyword, 589

bitfields, 592, 594

building an executable, 86

`__bytereverse` keyword, 588

compiler driver, 85

compiler limitations, 604

const keyword, 590

enum data type, 595

extended characters, 600

GNU dialect, 573, 576

Green Hills implementation of, 564

ISO C99 compliance, 579

Japanese Automotive C, 587

K & R dialect, 582 to 583

Kernighan & Ritchie conformance, 194

`__littleendian` keyword, 589

MISRA C, 587

MISRA C 1998, 216

MISRA C 2004, 197

`__packed` keyword, 589

preprocessing, 600

reentrant functions in run-time

libraries, 714

run-time libraries, 704

specifying a dialect of, 564

standard libraries, 704

<stdarg.h>, 597

Strict ANSI dialect, 572

struct data type, 591

structures, 771

`__thread` keyword, 590

thread-local storage, 590

type qualifiers, 588

union data type, 591

<varargs.h>, 596

variable arguments, 595

volatile keyword, 590

C Language Dialect Builder option, 194,
564 to 565, 572 to 573, 579, 582, 587 to
588, 590, 592, 600

C++, *see* C++ language

C++ Exception Handling Builder
option, 196, 623 to 624

C++ Include Directories Builder
option, 124, 302

C++ Inlining Level Builder option, 303,
328

C++ language

alignment and size limitations, 777

ARM compliant, 630

building an executable, 86

Index

- C `main()`, with, 765
- compiler driver, 85
- data types, 776
- decode** utility, 656
- Embedded dialect, 627
- Extended Embedded dialect, 629
- extensions, 624
- GNU dialect, 631
- implementation-defined features, 656
- in C programs, 767
- inlining, 328
- International Standard, 624
- linkage, 654
- namespaces, 645
- post processing, 655
- precompiled header files, 648 to 653
- predefined macros for, 666
- required macro names, 664
- specifying a dialect of, 622
- Standard, 624
 - See also* Standard C++
- standard libraries, 704
- templates, 234 to 238, 634 to 636, 638, 640 to 641
- C++ Language Dialect** Builder
 - option, 195 to 196, 622 to 624, 627, 629 to 631
- C++ Libraries** Builder option, 196, 623
- C++ library, specifying, 623
- C++ Linking Method** Builder option, 308
- C++ Programming Language, Third Edition, The*, 626
- C++ Standard Include Directories**
 - Builder option, 302
- c_and_cpp_functions_are_distinct**
 - driver option, 308
- C/C++ minimum/maximum
 - optimization, 359
- C/C++ Minimum/Maximum Optimization** Builder option, 290, 359
- c_include_directory** driver option, 302
- __c_plusplus**, 666
- c99** driver option, 194, 564
- C99** driver option, 194, 564
- Call Graph Generation** Builder
 - option, 257
- call graph profiling
 - debugging information, 118
- call_shared** driver option, 162
- callgraph** driver option, 257
- calling conventions, 7, 107
- `calloc()`, ANSI C implementation, 618
- .cc** file extension, 40, 87
- ccppc** compiler driver, 85
- `cerr`, 766
- `char` data type, 106
 - signedness, 225, 670, 677
 - size, 669
 - variable arguments, 596
- __CHAR_BIT**, 669
- __Char_Is_Signed__**, 670, 677
- __Char_Is_Unsigned**, 677
- __Char_Is_Unsigned__**, 670
- __CHAR_UNSIGNED__**, 670
- character constants, 398
- character encoding, 105
- character set dependencies, 780
- check** driver option, 190 to 191
- check=all** driver option, 189
- check=alloc** driver option, 191
- check=assignbound** driver option, 189
- check=bounds** driver option, 189
- check=memory** driver option, 191
- check=nilderef** driver option, 189
- check=nomemory** driver option, 191
- check=none** driver option, 189
- check=switch** driver option, 189
- check=watch** driver option, 190
- check=zerodivide** driver option, 189
- Checkout Browser** Project Manager menu
 - item, 78
- Checksum** Builder option, 150, 258

Index

- checksum** driver option, 258
- cin**, 766
- classes**, 778
 - anonymous, 625
 - portability, 778
 - template, 634
- Clean all output before building** Project Manager menu item, 75
- Clean *selected file*** Project Manager menu item, 74
- CLEAR**
 - linker section attribute, 452
- Clear User Default Configuration** Project Manager menu item, 78
- Close Project Manager** Project Manager menu item, 67
- Close Project** Project Manager menu item, 66
- __CLZ32 ()** intrinsic function, 697
- Code Factoring** Builder option, 251, 469
- code factoring optimization**
 - controlling the scope of, 469
 - incremental linking, 470
 - invoking, 469
 - overview, 469
 - partial, 470
- codefactor** driver option, 251, 469
- __COFF__**, 675
- collating sequence**, 781
- .comm** assembler directive, 426, 774
- COMMAND** context sensitive variable, customization file, 859
- command help**, 10
- command line**
 - for assembler, 394
 - for **gdump** utility, 411
- command line utilities**, *see* utility programs
- command=command_name .gpj**
 - conditional control statement, 838
- commands**
 - help regarding, 10
 - helpview**, 10
- Commands to Execute After Associated Command (via Shell) Builder**
 - option, 282
- Commands to Execute After Associated Command** Builder option, 281
- Commands to Execute Before Associated Command (via Shell) Builder**
 - option, 280
- Commands to Execute Before Associated Command** Builder option, 279
- comment .gpj** conditional control statement, 838
- comments**, in assembler source, 399
- Common Subexpression Elimination** Builder option, 289, 355
- common subexpression elimination optimization**, 355
- commons** driver option, 227
- Compile *selected file*** Project Manager menu item, 73
- compiler driver**, 84
 - See also* driver options
 - building a C executable with, 86
 - building a C++ executable with, 86
 - ccppc**, 85
 - controlling the assembler with, 125
 - controlling the linker with, 126
 - creating libraries with, 91
 - cxppc**, 85
 - file extensions recognized by, 87
 - for C source files, 85
 - for C++ source files, 85
 - generating assembly files with, 90
 - generating debugging information with, 115
 - generating object files with, 90
 - input file types, 87
 - introduction to, 7
 - linker directives files, passing to, 88
 - optimization, *see* optimizations

Index

- optimizing your code with, 176
- output file types, 90
- passing multiple files to, 88
- renaming output of, 147
- running assembler with, 392
- setting optimization options for, 176
- specifying a target for, 113
- syntax, 85
- updating libraries with, 91
- using a driver options file with, 95
- using files with, 85
- using options with, 85
- .con** file extension, 41
- conditional assembly directives, 416
- conditional control statements
 - optgroup=option_group**, 839
- conditional control statements, for **.gpj** files, 837, 840
- configuration options
 - help regarding, 10
- Configuration** Project Manager menu item, 78
- Configure** Project Manager menu item, 67
- configuring existing items
 - Project Manager, 33
- Connect** Project Manager menu item, 76
- Connection Organizer** Project Manager menu item, 76
- Consistent code without debug information** Builder option, 293
- consistentcode** driver option, 293
- const** keyword, 583, 590
- constant data in ROM, 458
- constant folding optimization, 364
- Constant Propagation** Builder option, 290, 358
- constant propagation optimization, 358
- constant returns, propagation of
 - interprocedural optimizations, 331
- constants, in assembler, 397
- constructors, 655, 765, 768
- context-sensitive help, 10
- continuation lines, in assembler, 400
- Contract Project** Project Manager menu item, 72
- conventions
 - typographical, xxii
- Convert DWARF Files to MULTI (.dbo)** **Format** Builder option, 298
- Convert Legacy project** Project Manager menu item, 79
- Convert Stabs Files to MULTI (.dbo)** **Format** Builder option, 299
- Copy selected file as Link** Project Manager menu item, 69
- Copy selected file Local** Project Manager menu item, 69
- copyright banner, generating, 260
- cout**, 766
- coverage profiling
 - debugging information, 119
- __cplusplus**, 665
- .cpp** file extension, 40, 87
- cpu** assembler option, 394
- cpu=** driver option, 830
- cpu=** Green Hills GNU option, 830
- cpu=cpu** driver option, 113 to 114, 269
- CRC, 150, 258, 487
- Create Auto-Include Subproject** Project Manager menu item, 70
- Create Raw Binary Image** Builder option, 169
- Create RPL/RPX** Builder option, 163
- create_pch** driver option, 648
- create_rpl** driver option, 163
- create_rpx** driver option, 163
- creating a library
 - Project Manager, 474
- CROM linker section attribute, 453, 456
- cross compilers, 7
- Cross-Reference Information** Builder option, 297

Index

crt0.o startup module, 459, 753

crt0.ppc, 753

customization files (**.bod**), 844

 CommandOptions

 definitions, 847

 syntax, 861

 Commands

 definitions, 846

 syntax, 861

components, 844

context sensitive variables

 COMMAND, 859

 DISKFILE, 859

 EDITOR, 859

 FILENAME, 859

 FILENAMEBASE, 859

 FILENAMESLIBBASE, 859

 FILETYPEOPTIONS, 859

 INPUTDIR, 859

 INPUTFILE, 859

 ITEMID, 859

 KEEPOBJECTS, 859

 MAKE_DEPEND_OPTIONS, 859

 OBJECTBASE, 860

 OBJECTS, 859

 OBJECTSUFFIX, 860

 OPTIONS, 860

 OUTPUTDIR, 860

 OUTPUTFILE, 860

 OUTPUTNAMEBASE, 860

 PARENTID, 860

 RUNDIR, 860

 SECOND_PASS_OPTION, 860

 SIBLINGS, 860

 SIBLINGS[*type*], 860

 syntax, 858

 TOPPROJECT, 860

examples, 849

FileTypes

 definitions, 845

 syntax, 856

including additional, 864

predefined macro syntax, 864

preprocessor directives

 #define, 863

 #echo, 863

 #elif, 863

 #else, 863

 #elseif, 863

 #elif, 863

 #endif, 863

 #error, 863

 #if, 863

 #ifdef, 863

 #include, 863

 #info, 863

 syntax, 863

 #warning, 863

preprocessor function syntax, 863

customization Top Project directive, 836

Customize Menus Project Manager menu

 item, 79

customizing

 run-time environment, 146

cxppc compiler driver, 85

.cxx file extension, 40, 87

--cxx_include_directory driver

 option, 124, 302

cxxmunch utility, 655

D

-D driver option, 192

-d librarian command, 476

.d output file type, 93, 101

 .data assembler directive, 421

data initialization assembler directives, 417

 .data program section, 127, 458, 813

data structures

 thread-specific library, 717

data type

 alignment requirement, 106

Index

- size, 106
- data types
 - size, 776
 - void, 583
- data_delete** driver option, 250
- `__DATE__`, 583, 665
 - precompiled header files and, 651
- .dba** output file type, 93
- dbg_source_root** driver option, 172
- Dbo File Version** Builder option, 314
- .dbo** output file type, 93
- dbo_trace** driver option, 313
- dbo_version** driver option, 314
- dbopath** driver option, 295
 - `__DCBF()` intrinsic function, 696
 - `__DCBST()` intrinsic function, 696
 - `__DCBT()` intrinsic function, 696
 - `__DCBZ()` intrinsic function, 696
- dead code elimination optimization, 360
- Debug Information (.dbo) Tracing**
 - Diagnostics** Builder option, 313
- Debug Other Executable** Project Manager
 - menu item, 77
- Debug selected program** Project Manager
 - menu item, 77
- debugger
 - displaying memory-mapped I/O, 149
- debugging
 - source level, 785
 - symbolic, 427
- debugging information
 - call graph profiling, 118
 - controlling level of, 298
 - coverage profiling, 119
 - files containing, 116
 - generating, 115
 - generating DWARF, 298
 - maintaining, 116
 - obtaining profiling, 117
 - target-based timing profiling, 120
- Debugging Information** Builder
 - option, 298
- Debugging Level** Builder option, 115 to 116, 187, 288, 298, 690
- decode** utility, 656
- defaultBuildOptions** Top Project
 - directive, 836
- DEFAULTS** linker directive, 439
- defer_parse_function_templates** driver option, 240
- Deferral of Function Template Parsing**
 - Builder option, 239, 304
- Define Linker Constant** Builder
 - option, 253
- %define** preprocessor directive,
 - customization file, 863
- Define Preprocessor Symbol** Builder
 - option, 192
- `#defined(id)`, 583
- Definition of Standard Symbols** Builder
 - option, 300
- Definition of Unsafe Symbols** Builder
 - option, 300, 664
- Delay Archives** Builder option, 292
- Delay Second Pass** Builder option, 292
- delay_archives** driver option, 292
- delay_second_pass** driver option, 292
- delete
 - array, predefined macro, 675
- delete** driver option, 250
- Delete selected file** Project Manager menu item, 69
- Delete Unused Kernel Calls** Builder
 - option, 169
- deleting
 - unused functions, 467
- Deletion of Unused C++ Virtual Functions** Builder option, 250
- Deletion of Unused Data** Builder
 - option, 250

Index

Deletion of Unused Functions Builder
option, 126, 250

demangler, 656

Demangling of Names in Linker

Messages Builder option, 308

.dep output file type, 93, 101

--dep_name driver option, 238, 624, 641, 645

Dependencies Relative to Source

Directory Builder option, 277

Dependencies Relative to This File

Builder option, 277

Dependencies Relative to Top-Level

Project Builder option, 277

dependency information, generating, 100

Dependent Name Processing Builder

option, 238 to 239, 304, 624, 641, 645

:depends Builder option, 277

:dependsNonRelative Builder option, 277

:dependsRelative Builder option, 277

derefed DoubleCheck property
type, 371

destructors, 655, 765, 768

__DI() intrinsic function, 693

--diag_error driver option, 264

--diag_remark driver option, 265

--diag_suppress driver option, 265

--diag_warning driver option, 265

__DIR() intrinsic function, 694

-directive_dir driver option, 310

directories

library, 705

--disable_ctors_dtors driver option, 232

--disable_noinline driver option, 231, 329

-discard_zero_initializers driver
option, 159

DISKFILE context sensitive variable,
customization file, 859

Display BootTable Builder option, 169

Display Error Message Numbers Builder
option, 264 to 265, 686

Display Error Message Paths Builder
option, 261

Display includes Preprocessor Directives
Listing Builder option, 192

Display Local Symbols in Map File
Builder option, 256

Display Unused Functions in Map File
Builder option, 256

Display Version Information Builder
option, 260

-display_boot intex option, 169

--display_error_number driver
option, 265, 686

-display_toc librarian option, 478

Distinct C and C++ Functions Builder
option, 308, 624

division

operators, 783

.dla output file type, 93

.dlo output file type, 93

.dnm output file type, 93

-do_not_wait_for_dblink driver
option, 297

document set, xx to xxi

-dotciscxx driver option, 88

.double assembler directive, 417

double data type, 106

-double_check.config= driver option, 267

-double_check.ignore= driver option, 267

-double_check.level=high driver
option, 266, 367

-double_check.level=low driver
option, 266, 367

-double_check.level=medium driver
option, 266, 367

-double_check.level=none driver
option, 266, 367

-double_check.report= driver
option, 266, 373

-double_check.stop_build=errors driver
option, 267, 374

Index

- double_check.stop_build=off** driver option, 267, 374
- double_check.stop_build=warnings** driver option, 267, 374
- __DOUBLE_HL**, 671
- DoubleCheck**
 - controlling output, 374
 - custom functions, using, 368
 - error and warning messages, 374 to 376
 - function properties, specifying
 - with #pragma directives, 370
 - with configuration files, 369
 - overview, 366
 - property types
 - derefed, 371
 - frees, 371
 - malloc_size, 371
 - malloced, 370
 - needs_null_check, 370
 - no_return, 372
 - no_return_when_non_zero, 372
 - no_return_when_zero, 371
 - unsaved, 372
 - specifying report files, 367
 - using, 367
 - viewing reports, 373
- DoubleCheck Config File Builder** option, 267
- DoubleCheck Errors to Ignore Builder** option, 267
- DoubleCheck Level Builder** option, 266
- DoubleCheck Output that Stops Builds** Builder option, 267
- DoubleCheck Report File Builder** option, 266
- driver options
 - #**, 95, 830
 - a**, 119, 188
 - A**, 254
 - additional_sda_reg**, 145, 269
 - allow_different_section_types**, 311
 - alternative_tokens**, 229
 - Altivec**, 830
 - ansi**, 194, 564 to 565, 587
 - ANSI**, 194, 564, 572, 588
 - ansi_alias**, 197
 - any_output_suffix**, 275
 - archive**, 91 to 92, 474
 - arm**, 195, 630
 - asm**, 242, 392
 - asm_errors**, 262, 599
 - asm_silent**, 262, 588, 599
 - asm_warnings**, 262, 599
 - asm3g**, 243
 - asmcmd**, 243, 392
 - assembler_warnings**, 309
 - auto_instantiation**, 239
 - auto_sda**, 141, 146, 158
 - :autoInclude**, 285
 - autoregister**, 289
 - bigendian**, 154, 830
 - bigswitch**, 273
 - bool**, 230, 675
 - brief_diagnostics**, 260
 - bsp**, 113
 - c**, 90, 92, 238
 - C**, 253, 301
 - c_and_cpp_functions_are_distinct**, 308
 - c_include_directory**, 302
 - c99**, 194, 564
 - C99**, 194, 564
 - call_shared**, 162
 - callgraph**, 257
 - check**, 190 to 191
 - check=all**, 189
 - check=alloc**, 191
 - check=assignbound**, 189
 - check=bounds**, 189
 - check=memory**, 191
 - check=nilderef**, 189
 - check=nomemory**, 191

Index

- check=none, 189
- check=switch, 189
- check=watch, 190
- check=zerodivide, 189
- checksum, 258
- codefactor, 251, 469
- commons, 227
- consistentcode, 293
- cpu=, 830
- cpu=cpu, 113 to 114, 269
- create_pch, 648
- create_rpl, 163
- create_rpx, 163
- cxx_include_directory, 124, 302
- D, 192
- data_delete, 250
- dbg_source_root, 172
- dbo_trace, 313
- dbo_version, 314
- dbopath, 295
- defer_parse_function_templates, 240
- delay_archives, 292
- delay_second_pass, 292
- delete, 250
- dep_name, 238, 624, 641, 645
- diag_error, 264
- diag_remark, 265
- diag_suppress, 265
- diag_warning, 265
- directive_dir, 310
- disable_ctors_dtors, 232
- disable_noinline, 231, 329
- discard_zero_initializers, 159
- display_error_number, 265, 686
- do_not_wait_for_dblink, 297
- dotciscxx, 88
- double_check.config=, 267
- double_check.ignore=, 267
- double_check.level=high, 266, 367
- double_check.level=low, 266, 367
- double_check.level=medium, 266, 367
- double_check.level=none, 266, 367
- double_check.report=, 266, 373
- double_check.stop_build=errors, 267, 374
- double_check.stop_build=off, 267, 374
- double_check.stop_build=warnings, 267, 374
- dual_debug, 298
- dwarf2dbo, 298
- dyload, 168
- e, 147, 245
- e, 195, 627, 666
- E, 92, 238, 600
- e500_inst_fix, 270
- e500_inst_fix2, 271
- ee, 195, 629
- eel, 196, 667, 706
- eele, 196, 667, 706
- eieio_loads, 161
- eieio_stores, 161
- el, 196, 667, 706
- ele, 196, 667, 706
- enable_ctors_dtors, 232
- enable_noinline, 231, 329
- endfile, 252
- errmax, 259
- error_basename, 261
- event_logging, 166
- exceptions, 196, 624
- export, 238, 624, 641
- extend_liveness, 294
- external, 181
- external_file, 181
- extra_file=filename, 296
- extractall, 254
- farcallpatch, 159
- farcalls, 136, 158, 830
- fast, 185

Index

- fhard, 155
- @file, 95
- filetype.suffix, 88
- float_scanf, 310
- floatio, 157
- floatsingle, 156
- fNaN_cmp_unordered, 156
- fno_NaN_cmp_unordered, 156
- fnone, 155, 706
- fnoprecise, 156
- follow_section, 312
- force_uvfd, 311
- force_vtbl, 234
- fprecise, 156
- fsoft, 155
- full_breakdots, 294
- full_debug_info, 298
- g, 187
- G, 115 to 116, 187, 707
- g++, 195, 631
- ga, 295
- gcc, 194, 564, 573
- glimits, 293
- globalreg, 134, 146, 160, 673
- gnu_asm, 305
- gnu_version=30300, 303
- gnu_version=40300, 304
- gnu_version=40300_strict, 304
- gs, 187
- gsize, 257
- gtws, 293
- guiding_decls, 235
- H, 101, 192
- help, 95
- hex, 245
- I, 122, 172
- I-, 123
- ident, 309
- identoutput, 309
- ignore_debug_references, 252
- implicit_extern_c_type_conversion, 308
- implicit_include, 236, 640
- implicit_typename, 236
- include, 300
- include_never, 299
- include_once, 299
- incorrect_pragma_errors, 262, 679
- incorrect_pragma_silent, 262, 679
- incorrect_pragma_warnings, 262
- inline_prologue, 273
- inline_tiny_functions, 288
- inline_trivial, 251
- inlining, 303, 328
- inlining_unless_debug, 303, 328
- instantiate_extern_inline, 231
- instantiation_dir, 237
- integrity_posix, 168
- :integrity_posix_mode=noposix, 168
- :integrity_posix_mode=single, 168
- :integrity_posix_mode=system, 168
- interrupt_prologue_always_uses_stm, 272
- irel, 167
- irelprog, 167
- isel, 270
- japanese_automotive_c, 195, 587 to 588
- k+r, 194, 564, 582
- kanji, 229, 603
- keep_static_symbols, 228
- keepmap, 255
- keeptempfiles, 275
- kernel, 166
- kernel=kernel_checked, 166
- kernel=kernel_opt, 166
- kernel=kernel_vel, 166
- kernel=kernel_vel_checked, 166
- kernel=kernel_vel_opt, 166
- l, 125, 173
- L, 125, 173
- language, 275
- libbsp, 167
- link_constprop, 251

Index

--link_filter, 308
--link_once_templates, 235, 638
--link_output_mode_linksafe, 311
--link_output_mode_reuse, 311
--link_output_mode_safe, 311
--link_output_mode_unlink, 311
--linker_link, 308
-linker_warnings, 246
-list, 125, 241
-list_dir, 241
-littleendian, 154, 830
-lnk, 248
-lnk0, 248
-lnkcmd, 249
-locals_unchanged_by_longjmp, 304
-locatedprogram, 244
--long_long, 226
-Ma, 256
-Make, 92
-makerpl.args, 165
-map, 255
-maplines, 255
--max_inlining, 303, 328
--max_inlining_unless_debug, 303, 328
-MD, 101
-memory, 245
-merge_archive, 92, 475
--misra_2004=1.1, 200
--misra_2004=1.2, 200
--misra_2004=1.3, 200
--misra_2004=1.4, 200
--misra_2004=1.5, 200
--misra_2004=10.1, 205
--misra_2004=10.2, 205
--misra_2004=10.3, 205
--misra_2004=10.4, 205
--misra_2004=10.5, 206
--misra_2004=10.6, 206
--misra_2004=11.1, 206
--misra_2004=11.2, 206
--misra_2004=11.3, 206
--misra_2004=11.4, 206
--misra_2004=11.5, 206
--misra_2004=12.1, 206
--misra_2004=12.10, 207
--misra_2004=12.11, 207
--misra_2004=12.12, 208
--misra_2004=12.13, 208
--misra_2004=12.2, 206
--misra_2004=12.3, 207
--misra_2004=12.4, 207
--misra_2004=12.5, 207
--misra_2004=12.6, 207
--misra_2004=12.7, 207
--misra_2004=12.8, 207
--misra_2004=12.9, 207
--misra_2004=13.1, 208
--misra_2004=13.2, 208
--misra_2004=13.3, 208
--misra_2004=13.4, 208
--misra_2004=13.5, 208
--misra_2004=13.6, 208
--misra_2004=13.7, 208
--misra_2004=14.1, 209
--misra_2004=14.10, 209
--misra_2004=14.2, 209
--misra_2004=14.3, 209
--misra_2004=14.4, 209
--misra_2004=14.5, 209
--misra_2004=14.6, 209
--misra_2004=14.7, 209
--misra_2004=14.8, 209
--misra_2004=14.9, 209
--misra_2004=15.1, 210
--misra_2004=15.2, 210
--misra_2004=15.3, 210
--misra_2004=15.4, 210
--misra_2004=15.5, 210
--misra_2004=16.1, 210
--misra_2004=16.10, 211
--misra_2004=16.2, 210

Index

--misra_2004=16.3, 210	--misra_2004=20.12, 215
--misra_2004=16.4, 211	--misra_2004=20.2, 214
--misra_2004=16.5, 211	--misra_2004=20.3, 214
--misra_2004=16.6, 211	--misra_2004=20.4, 214
--misra_2004=16.7, 211	--misra_2004=20.5, 214
--misra_2004=16.8, 211	--misra_2004=20.6, 214
--misra_2004=16.9, 211	--misra_2004=20.7, 215
--misra_2004=17.1, 211	--misra_2004=20.8, 215
--misra_2004=17.2, 211	--misra_2004=20.9, 215
--misra_2004=17.3, 211	--misra_2004=21.1, 215
--misra_2004=17.4, 211	--misra_2004=3.1, 201
--misra_2004=17.5, 212	--misra_2004=3.2, 201
--misra_2004=17.6, 212	--misra_2004=3.3, 201
--misra_2004=18.1, 212	--misra_2004=3.4, 201
--misra_2004=18.2, 212	--misra_2004=3.5, 201
--misra_2004=18.3, 212	--misra_2004=3.6, 201
--misra_2004=18.4, 212	--misra_2004=4.1, 201
--misra_2004=19.1, 212	--misra_2004=4.2, 202
--misra_2004=19.10, 213	--misra_2004=5.1, 202
--misra_2004=19.11, 213	--misra_2004=5.2, 202
--misra_2004=19.12, 213	--misra_2004=5.3, 202
--misra_2004=19.13, 213	--misra_2004=5.4, 202
--misra_2004=19.14, 213	--misra_2004=5.5, 202
--misra_2004=19.15, 214	--misra_2004=5.6, 202
--misra_2004=19.16, 214	--misra_2004=5.7, 202
--misra_2004=19.17, 214	--misra_2004=6.1, 203
--misra_2004=19.2, 212	--misra_2004=6.2, 203
--misra_2004=19.3, 212	--misra_2004=6.3, 203
--misra_2004=19.4, 212	--misra_2004=6.4, 203
--misra_2004=19.5, 212	--misra_2004=6.5, 203
--misra_2004=19.6, 213	--misra_2004=7.1, 203
--misra_2004=19.7, 213	--misra_2004=8.1, 203
--misra_2004=19.8, 213	--misra_2004=8.10, 204
--misra_2004=19.9, 213	--misra_2004=8.11, 204
--misra_2004=2.1, 200	--misra_2004=8.12, 205
--misra_2004=2.2, 201	--misra_2004=8.2, 203
--misra_2004=2.3, 201	--misra_2004=8.3, 204
--misra_2004=2.4, 201	--misra_2004=8.4, 204
--misra_2004=20.1, 214	--misra_2004=8.5, 204
--misra_2004=20.10, 215	--misra_2004=8.6, 204
--misra_2004=20.11, 215	--misra_2004=8.7, 204

Index

- misra_2004=8.8, 204
- misra_2004=8.9, 204
- misra_2004=9.1, 205
- misra_2004=9.2, 205
- misra_2004=9.3, 205
- misra_adv=error, 216
- misra_adv=silent, 216
- misra_adv=warn, 216
- misra_req=error, 215
- misra_req=silent, 215
- misra_req=warn, 215
- misra_runtime, 216
- MI, 256
- MMD, 101
- Mn, 256
- Mu, 256
- multi_version, 294
- multiple, 253
- munch, 308
- Mx, 256
- namespaces, 230
- new_inside_of_constructor, 233
- new_outside_of_constructor, 233
- new_style_casts, 232
- no_additional_output, 245
- no_alternative_tokens, 229
- no_ansi_alias, 197
- no_any_output_suffix, 275
- no_assembler_warnings, 309
- no_auto_instantiation, 239
- no_auto_sda, 158
- no_bool, 230
- no_brief_diagnostics, 261
- no_c_and_cpp_functions_are_distinct, 308
- no_callgraph, 257
- no_codefactor, 251
- no_comments, 301
- no_commons, 127, 228
- no_coverage_analysis, 188
- no_create_rpl, 163
- no_data_delete, 250
- no_debug, 187
- no_defer_parse_function_templates, 240
- no_delete, 250
- no_dep_name, 238
- no_discard_zero_initializers, 159
- no_display_error_number, 265
- no_dual_debug, 298
- no_dwarf2dbo, 298
- no_e500_inst_fix, 270
- no_e500_inst_fix2, 271
- no_eieio_loads, 161
- no_eieio_stores, 161
- no_error_basename, 261
- no_event_logging, 166
- no_exceptions, 196
- no_export, 238
- no_extend_liveness, 294
- no_float_scanf, 310
- no_full_breakdots, 294
- no_full_debug_info, 298
- no_gnu_asm, 305
- no_gsize, 257
- no_guiding_decls, 235
- no_ignore_debug_references, 252
- no_implicit_extern_c_type_conversion, 308
- no_implicit_include, 236, 641
- no_implicit_typename, 236
- no_include_once, 299
- no_inline_prologue, 273
- no_inline_tiny_functions, 288, 317
- no_inline_trivial, 251
- no_inlining, 303, 328
- no_instantiate_extern_inline, 232
- no_integrity_posix, 168
- no_interrupt_prologue_always_uses_stm, 272
- no_irel, 167
- no_irelprog, 167
- no_isel, 270
- no_japanese_automotive_c, 195
- no_keep_static_symbols, 228
- no_kernel, 166

Index

-no_libbsp, 167
--no_link_constprop, 251
--no_link_filter, 308
--no_link_once_templates, 235, 635
-no_linker_warnings, 246
-no_list, 241
-no_locals_unchanged_by_longjmp, 304
--no_long_long, 226
--no_misra_runtime, 216
-no_multiple, 253
--no_namespaces, 230, 645
--no_new_style_casts, 232
--no_old_specializations, 236
--no_one_instantiation_per_object, 237
-no_only_explicit_reg_use, 272
--no_parse_templates, 238
--no_pch_match_directory, 649
--no_pch_messages, 652
-no_posix_dll, 168
-no_precise_signed_zero, 157
--no_prelink_objects, 238
-no_preprocess_assembly_files, 241
--no_preprocess_linker_directive, 246
-no_preprocess_special_assembly_files, 242
-no_preprpl_all_libs, 164
--no_quit_after_warnings, 260
--no_readonly_typeinfo, 233
--no_readonly_virtual_tables, 234
--no_remarks, 259
--no_restrict, 231
-no_rpl_use_dbg_source_root, 164
--no_rtti, 233
--no_scan_source, 296
-no_search_for_dba, 295
--no_search_source_directory, 303
--no_short_enum, 226, 588
--no_slash_comment, 196
-no_stabs2dbo, 299
-no_stack_check, 160
-no_timer_profile, 188
--no_trace_includes, 193
-no_undefined, 253
--no_unique_strings, 228
--no_unsafe_predefines, 300
-no_use_fsel, 270
--no_using_std, 230
-no_uvfd, 251
--no_version, 260
--no_wrap_diagnostics, 261
-noAltivec, 830
-noasm3g, 243
-noautoregister, 289, 363
-nobigswitch, 273
-nochecksum, 258
-noconsistentcode, 293
--nocpp, 308
-nodelay_archives, 292
-nodelay_second_pass, 292
-noentry, 245
-nofarcallpatch, 135, 159
-nofarcalls, 159, 830
-nofloatio, 157, 706
-nofloatsingle, 156
-noga, 296
-noglimits, 293
-nogtws, 293
-noidentoutput, 309
-nokeepmap, 255
-nokeeptempfiles, 275
-nomap, 255
-non_shared, 162
-noobj, 274
-nooverlap, 257
-nooverload, 289, 362
-nopackage_infs, 292
-nopasssource, 242
-noSPE, 271
-nostartfiles, 252
-nostddef, 300
-nostdinc, 302
-nostdlib, 310, 706
-nostrip, 245

Index

- nothreshold, 158
- novle, 162
- o, 91, 147, 174
- O, 176
- O1, 291
- O2, 291
- O3, 292
- OB, 180, 324
- obj, 274
- object_dir, 172
- Oconstprop, 290, 358
- Ocse, 289, 355
- Odebug, 176, 351
- Ogeneral, 176, 316 to 317, 351
- OI, 178, 186, 316, 321 to 322
- Ointerproc, 179, 330, 339
- Oip_analysis_only, 179
- Oipaliaslibfuncs, 185
- Oipaliasreads, 184
- Oipaliaswrites, 184
- Oipconstantreturns, 184
- Oipconstglobals, 183
- Oipconstprop, 183
- Oipdeletefunctions, 182
- Oipdeleteglobals, 182
- Oiplimitinlining, 184
- Oiponesiteinlining, 182, 335
- Oipremoveparams, 183
- Oipremovereturns, 184
- Oipsmallinlining, 183
- OL, 186, 291, 339 to 340, 785
- old_specializations, 236
- Olimit, 351 to 352
- Olimit=peephole, 180
- Olimit=pipeline, 180
- Olink, 179
- OM, 290, 359, 590, 784 to 785
- Omax, 180
- Omaxdebug, 176
- Omemfuncs, 288
- Ominmax, 290, 359
- Omoredebug, 176, 351
- one_instantiation_per_object, 237, 637
- only_explicit_reg_use, 272
- Onobig, 180
- Onoconstprop, 290, 358
- Onocse, 289, 355
- Onoinline, 178, 321
- Onoipa, 179, 337
- Onoipaliaslibfuncs, 185, 331
- Onoipaliasreads, 184, 330
- Onoipaliaswrites, 185, 331
- Onoipconstantreturns, 184, 331
- Onoipconstglobals, 183, 336
- Onoipconstprop, 183, 335
- Onoipdeletefunctions, 182, 336
- Onoipdeleteglobals, 182, 336
- Onoiplimitinlining, 184
- Onoiponesiteinlining, 182, 335
- Onoipremoveparams, 183, 335
- Onoipremovereturns, 184, 335
- Onoipsmallinlining, 183, 331
- Onolink, 179
- Onoloop, 291, 339
- Onomax, 180
- Onomemfuncs, 288, 326
- Onomemory, 290, 359, 785
- Onominmax, 290, 359
- Onone, 177, 317
- Onopeephole, 291, 351
- Onopipeline, 291, 352
- Onoprintfuncs, 288
- Onostrfuncs, 288, 326
- Onotailrecursion, 290, 356
- Onounroll, 289, 339, 342
- Onounrollbig, 180
- Onovector, 178
- Onovectorpeel, 292, 350
- Opeep, 291, 351
- Opipeline, 291, 352
- Oprintfuncs, 288

Index

--option, 313
:optionsFile, 286
-os_dir, 163
-Osize, 177, 273, 316 to 317, 329, 351
-Ospace, *see* driver options, **-Osize**
-Ospeed, 177, 316 to 317, 351
-Ostrfuncs, 288
-Otailrecursion, 290, 356
-Ounroll, 289
-Ounrollbig, 180, 342
-OV, 178, 344
-Ovectorpeel, 292
-overlap, 257
-overlap_warn, 257
-overload, 289
-Owholeprogram, 179, 334, 336, 339
-p, 117, 187
-P, 92, 238, 600
-pack, 227
-package_infs, 292
-paddr_offset, 312
--parse_templates, 238
-passsource, 242
--pch, 649, 652
--pch_dir, 648, 652
-pg, 117, 187
-pnone, 187
--posix_dll, 168
--pragma_message, 301
--pragma_message_silent, 301
--pragma_message_unknown, 301
-precise_signed_zero, 157
-prelink_against, 239
--prelink_objects, 238
-preprocess_assembly_files, 241, 392
--preprocess_linker_directive, 246
--preprocess_linker_directive_full, 246
-preprocess_special_assembly_files, 242
-preprpl_all_libs, 164
-preprpl.args, 165
-preprpl_output, 164
--prototype_errors, 261
--prototype_silent, 261
--prototype_warnings, 261
-Q, 92
-Qn, 246
--quit_after_warnings, 260
-Qy, 246
-rawimport, 246
--readonly_typeinfo, 233
--readonly_virtual_tables, 234
:reference, 285
--register_definition_file, 154
-relobj, 244
-relprog, 244
-relprog_cafe, 244
--remarks, 259
--restrict, 231
-rpl_use_dbg_source_root, 163
--rtti, 233
-S, 90, 92, 238, 393
--saferc, 217 to 225
--scan_source, 296
-sda, 138, 158
-sda=0, 830
-search_for_dba, 295
--search_source_directory, 302
-section, 161
--section_prefix, 304
--section_suffix, 304
--short_enum, 107, 226, 595
--signed_chars, 225, 609
--signed_fields, 225, 592
--signed_pointer, 226
--slash_comment, 196
--small, 185
-SPE, 271
-srec, 245
-stabs2dbo, 299
-stack_check, 160
--standard_vtbl, 234
-startfile_dir, 253

Index

- startfiles, 252
- std, 195, 624
- STD, 195, 624
- std_cxx_include_directory, 302
- stderr, 260
- stderr_overwrite, 312
- stdinc, 302
- stdl, 196, 667
- stdle, 196, 667, 706
- stdlib, 310
- strict_overlap_check, 257
- strip, 245
- suppress_vtbl, 234
- syntax, 92
- sys_include_directory, 302
- syslib, 162
- T, 247
- timer_profile, 121, 188
- tmp, 274
- u, 254
- U, 192
- UA, 255
- undefined, 253
- unique_strings, 228
- unknown_pragma_errors, 262, 679
- unknown_pragma_silent, 262, 679
- unknown_pragma_warnings, 262
- unsafe_predefines, 300, 664
- unsigned_chars, 225, 587, 609
- unsigned_fields, 225, 587, 592
- unsigned_pointer, 226
- use_fsel, 270
- use_pch, 648
- using_std, 230, 675
- uvfd, 251
- v, 95
- V, 260
- vle, 162
- w, 259, 607
- wait_for_dblink, 297
- warn_dbo_not_found_max, 296
- warnings, 259
- Wformat, 263
- Wimplicit-int, 263
- Wl, 249
- Wno-format, 263
- Wno-implicit-int, 263
- Wno-shadow, 263
- Wno-trigraphs, 264
- Wno-undef, 264
- wrap_diagnostics, 261
- Wshadow, 263
- Wtrigraphs, 264
- Wundef, 264
- X, 313
- xref=declare, 297
- xref=full, 297
- xref=global, 297
- xref=none, 297
- Y0, 286
- Ya, 287
- Yl, 287
- YL, 286
- zda, 138, 158
- .dsect assembler directive, 426, 771
- dual_debug driver option, 298
- dwarf2dbo driver option, 298
- dyload driver option, 168
- Dynamic Download Project Builder**
 - option, 168
- dynamic initialization, 655
- dynamic_intex option, 169

E

- e driver option, 147, 245
- e driver option, 195, 627, 666
- E driver option, 92, 238, 600
- e_ehsize ELF header field, 806
- e_entry ELF header field, 805
- e_flags ELF header field, 806
- e_ident ELF header field, 804

Index

- e_machine ELF header field, 805
- e_phentsize ELF header field, 806, 822
- e_phnum ELF header field, 806, 822
- e_phoff ELF header field, 806
- e_shentsize ELF header field, 806, 808
- e_shnum ELF header field, 806, 808
- e_shoff ELF header field, 806, 808
- e_shstrndx ELF header field, 806, 821
- e_type ELF header field, 804
- e_version ELF header field, 805
- e500_inst_fix driver option, 270
- e500_inst_fix2 driver option, 271
- EABI, PowerPC, 104, 107
- earlyabsolute linker option, 434
- EB legacy GNU option, 830
- EC++, *see* Embedded C++
- %echo preprocessor directive, customization file, 863
- ecxx header file directory, 705
- EDG Front End Options** Builder option, 313
 - __EDG__, 667
 - __EDG_IMPLICIT_USING_STD, 675
 - __EDG_RUNTIME_USES_NAMESPACES, 675
- Edit** Project Manager menu item, 68
- EDITOR context sensitive variable, customization file, 859
- Editor** Project Manager menu item, 78
- ee driver option, 195, 629, 667
- EEC++, *see* Extended Embedded C++
- ecxx header file directory, 705
- eel driver option, 196, 667, 706
- eele driver option, 196, 667, 706
 - __EFP_APU__, 668
 - __EI() intrinsic function, 693
- EI_CLASS ELF identification index, 807
- EI_DATA ELF identification index, 807 to 808
- EI_MAGn ELF identification index, 807
- EI_NIDENT ELF identification index, 807 to 808
- EI_PAD ELF identification index, 807 to 808
- EI_VERSION ELF identification index, 807 to 808
- __EIEIO() intrinsic function, 693
- eieio_loads driver option, 161
- eieio_stores driver option, 161
- __EIR() intrinsic function, 694
- .eject assembler directive, 424
- el driver option, 196, 667, 706
- EL legacy GNU option, 830
- ele driver option, 196, 667, 706
- ELF file format
 - data types, 803
 - header fields, 804
 - identification indexes, 807
 - introduction, 802
 - object and executable file formats, 802, 815
 - optional header output, 822
 - program attribute flags, 824
 - program section names, 812
 - program sections, 812
 - program types, 823
 - relocation directories, 815
 - relocation fields, 816
 - section attribute flags, 811
 - section header fields, 808
 - section types, 811
 - special section indexes, 810
 - string tables, 821
 - symbol bindings, 819
 - symbol table entries, 818
 - symbol types, 819
 - symbol values, 820
- ELF files
 - printing with **gdump** utility, 411
 - __ELF__, 675
 - #elif, 583

Index

- %elif** preprocessor directive, customization file, 863
- .else** assembler directive, 416
- %else** preprocessor directive, customization file, 863
- .elseif** assembler directive, 416
- %elseif** preprocessor directive, customization file, 863
- %elsif** preprocessor directive, customization file, 863
- elxr**, *see* linker
- Embedded C++
 - features of, 627
 - predefined macro, 666
- embedded development
 - and memory-mapped I/O, 148
 - and multiple-section programs, 128
 - and ROM, 458
 - symbolic memory-mapped I/O, 148
- `__EMBEDDED_CXX`, 666
- `__EMBEDDED_CXX_HEADERS`, 667
- Emulated GNU Version** Builder
 - option, 303
- enable_ctors_dtors** driver option, 232
- enable_noinline** driver option, 231
- enable_noinlinedriver** option, 329
- enableCompression** HTML compiler
 - options, 268
- .end** assembler directive, 426, 771
- End Files** Builder option, 252
- endaddr** linker expression function, 449
- endfile** driver option, 252
- endianness
 - default, 104
 - portability problems, 776
 - predefined macros for, 669
 - specifying, 154, 394, 518, 544
- Endianness** Builder option, 154
- .endif** assembler directive, 416
- %endif** preprocessor directive, customization file, 863
- .endm** assembler directive, 420
- .endr** assembler directive, 421
- enum data type, 106, 583, 595
 - size, 226
- `__Enum_Field_Is_Signed__`, 670
- `__Enum_Field_Is_Unsigned__`, 670
- .equ** assembler directive, 401, 426
- errmax** driver option, 259
- errno()**, ANSI C implementation, 618
- #error**, 583
- .error** assembler directive, 428
- error linker expression function, 449
- error messages
 - in assembler listing, 428
 - treating warnings as, 260
- %error** preprocessor directive, customization file, 863
- error_basename** driver option, 261
- escape sequences, in assembler, 398
- establishing interrupt vectors
 - `#pragma intvect`, 148
- evaluation order, 782
- Event Logging** Builder option, 166
- event_logging** driver option, 166
- `__EXCEPTION_HANDLING`, 676
- exceptions, 630
 - predefined macro, 676
- `__EXCEPTIONS`, 676
- exceptions** driver option, 196, 624
- executable files, 244
 - ELF format, 802, 815
- Executable Stripping** Builder option, 245
- Exit All** Project Manager menu item, 67
- exit()**, ANSI C implementation, 618
- .exitm** assembler directive, 420
- Expand Project** Project Manager menu item, 72
- `%expand_path(relative_path)` **.gpj**
 - macro function, 841
- Expansion of e500 evmwhgs_a**
 - Intrinsics** Builder option, 270

Index

Expansion of e500 evmwhs __a_w

Intrinsics Builder option, 270

--export driver option, 238, 624, 641

expression **.gpj .gpj** conditional control statement, 839

expressions

assembler, 401

expressions, assembler, 406

Extend Variable Liveness Builder

option, 294

-extend_liveness driver option, 294

extended characters, 600

Extended Embedded C++

predefined macro, 667

__EXTENDED_EMBEDDED_CXX, 667

__EXTENDED_EMBEDDED_CXX_HEADERS, 667

.extern assembler directive, 426, 770

extern storage class specifier

``C'', 625

``C++'', 625

K & R C, 583

syntax, 654

use of, 654, 766

-external driver option, 181

-external_file driver option, 181

-extra_file=filename driver option, 296

-extractall

linker option, 466

-extractall driver option, 254

-extractall linker option, 434

-extractweak linker option, 434, 466

:extraOutputFile Builder option, 284

F

F1 key, for help, 10

__fabs() intrinsic function, 695

__fabsf() intrinsic function, 695

Far Calls Builder option, 136, 158

far function calls, 135

Builder option, 136

__farcall, 136

-farcallpatch driver option, 159

-farcalls driver option, 136, 158, 830

--fast driver option, 185

fenv.h

functions, 745

-fhard driver option, 155

__Field_Is_Signed__, 670

__Field_Is_Unsigned__, 670

.file assembler directive, 427

@file driver option, 95

file extension

.bsp, 41

.idl, 41

.int, 41

file extensions, 87

file inclusion directives, 418

File Names Only Project Manager menu

item, 72

file type=file type .gpj conditional control statement, 838

file types

Project Manager, 39

setting manually, 88

__FILE__, 583, 665

FILENAME context sensitive variable, customization file, 859

FILENAMEBASE context sensitive variable, customization file, 859

FILENAMESLIBBASE context sensitive variable, customization file, 859

files

searching for in project, 46

Files Listing External Symbols Builder

option, 181

Files to Pre-Include Builder option, 300

-filetype.suffix driver option, 88

FILETYPEOPTIONS context sensitive variable, customization file, 859

FILL linker section attribute, 452

Index

- Filter Project View** Project Manager menu item, 73
- final** linker expression function, 449
- .fixaddr** program section, 459, 814
- .fixtype** program section, 459, 814
- Flash *selected program*** Project Manager menu item, 78
- .float** assembler directive, 417
- float** data type, 106
 - variable arguments, 596
- <float.h>** header file, 776
- float_scanf** driver option, 310
- floating-point**
 - I/O, disabling, 157
 - language-independent library, 750
 - predefined macros for, 671 to 672
 - range, 781
 - values, 104
- Floating-Point Coprocessor** Builder option, 155
- Floating-Point Precise Signed Zero** Builder option, 156
- Floating-Point Truncation** Builder option, 156
- floatio** driver option, 157
- floatsingle** driver option, 156
 - __FMADD()** intrinsic function, 696
 - __FMADDS()** intrinsic function, 696
- fmod()**, ANSI C implementation, 615
 - __FMSUB()** intrinsic function, 696
 - __FMSUBS()** intrinsic function, 696
 - __fnabs()** intrinsic function, 695
 - __fnabsf()** intrinsic function, 695
- fNaN_cmp_unordered** driver option, 156
 - __FNMADD()** intrinsic function, 696
 - __FNMADDS()** intrinsic function, 696
 - __FNMSUB()** intrinsic function, 696
 - __FNMSUBS()** intrinsic function, 696
- fno_NaN_cmp_unordered** driver option, 156
- fnone** driver option, 155, 706
- fnoprecise** driver option, 156
- Follow Section** Builder option, 312
- follow_section** driver option, 312
- Force All Symbols From Library** Builder option, 254
- Force Deletion of Unused C++ Virtual Function Symbol** Builder option, 311
- Force Frame Pointer** Builder option, 295
- Force link** Project Manager menu item, 75
- Force Undefined Symbol** Builder option, 254
- force_uvfd** driver option, 311
- force_vtbl** driver option, 234
- fprecise** driver option, 156
- frees** DoubleCheck property type, 371
- __FRES()** intrinsic function, 696
- friend** classes, 625
 - __FRSQRT()** intrinsic function, 696
 - __FSEL()** intrinsic function, 696
- .fsize** assembler directive, 427
- fsoft** driver option, 155
- Full Breakdots** Builder option, 294
- Full Paths** Project Manager menu item, 72
- Full POSIX-2001 Compilation Support** Builder option, 168
- full_breakdots** driver option, 294
- full_debug_info** driver option, 298
 - __FULL_DIR__**, 674
 - __FULL_FILE__**, 674
 - __FUNCPTR_BIT**, 669
- function** declaration, far function call, 136
- Function Prologues** Builder option, 272, 685
- function** properties, specifying for
 - DoubleCheck, 369 to 370
 - __FUNCTION__**, 674
- functions**
 - aligning, 680
 - calling conventions, 779
 - compiler generated, 640

Index

- deleting unused, 467
- inline, 640, 680
- interrupt, 685
- keeping unused, 467
- prototype, C versus. C++, 768
- pure virtual, 640
- return values, 779
- variable argument, 595

Functions Without Prototypes Builder
option, 261, 263

G

- g** driver option, 187
- G** driver option, 115 to 116, 187, 707
- g++** driver option, 195, 631
- ga** driver option, 295
- gaddr2line** utility, 484
- gasmlist** utility, 485
- gbin2c** utility, 487
- gbincmp** utility, 496
- gbldconvert** utility, 498
- gbuild** utility, 499
 - Implicit Dependency Analysis (IDA), 505
 - using with interprocedural optimizations, 337
- gcc** driver option, 194, 564, 573
- gcolor** utility, 506
- gcompare** utility, 511
- gdump** utility, 411, 514
- .gen** assembler directive, 424
- Generate Additional Output Builder**
option, 92, 244
- Generate MULTI and Native Information Builder** option, 298
- Generate Target-Walkable Stack Builder**
option, 293
- Generated Header File Directory Builder**
option, 170
- Generated Header File Name Builder**
option, 170

-generatemonolithdeletions **intex**
option, 169

Generation of fsel Instructions Builder
option, 270

Generation of isel Instructions Builder
option, 270

getenv(), ANSI C implementation, 618

__GETSR() intrinsic function, 695

gfile utility, 521

gfunsize utility, 522

ghexfile utility, *see* **gsrec** utility

ghide utility, 523

__ghs__, 667

__ghs_alignment, 674

__ghs_asm, 667

__ghs_c_int__, 667

__ghs_fprecise__, 672

.ghs_info program section, 471

__GHS_Inline_Memory_Functions, 675

__GHS_Inline_String_Functions, 675

__ghs_max_pack_value, 674

__GHS_NOCOMMONS__, 673

__GHS_Optimize_Inline, 675

__ghs_packing, 674

__GHS_REVISION_DATE, 667

__GHS_REVISION_VALUE, 667

__ghs_sda, 673

__ghs_sda_threshold, 673

__GHS_VERSION_NUMBER, 667

__ghs_zda, 673

__ghs_zda_threshold, 673

__ghsalwaysimport

Linker Symbols, 466

__ghsautoimport

Linker Symbols, 466

__ghsbegin symbol, 463

__ghsend symbol, 463

.ghsnote assembler directive, 471

__ghssize symbol, 463

__ghstable *tablename*,

__ghsentry_tablename_entryname

Index

- Linker generated tables, 465
- glimits** driver option, 293
- .global** assembler directive, 426
- global objects, 655
- Global Registers** Builder option, 134, 146, 160
- global variables, 773
- globalreg** driver option, 134, 146, 160, 673
- __GlobalRegisters**, 673
- .globl** assembler directive, 426, 770
- gmemfile**
 - invoking from the driver, 244
- gmemfile** utility, 524
- .gmon** output file type, 93
- gnm** utility, 531, 655
- GNU C extensions, 573, 576
- GNU C++ extensions, 631
- GNU Compatibility Optimizations**
 - Builder option, 291
- __gnu_asm**, 668
- gnu_asm** driver option, 305
- gnu_version=30300** driver option, 303
- gnu_version=40300** driver option, 304
- gnu_version=40300_strict** driver option, 304
- __GNUC__**, 666
- .gpj** file extension, 20, 39
 - See also* project files
- .gpj** syntax, advanced, 841
 - conditional control statements
 - command=command_name**, 838
 - comment**, 838
 - expression*, 839
 - file type=file type**, 838
 - nobuild**, 839
 - optional**, 840
 - overview of, 837
 - pass=**, 840
 - rebuild**, 840
 - selectone_optional**, 840
 - selectone_required**, 840
 - target!=target_name**, 840
 - target=target_name**, 840
 - macro functions
 - %expand_path(relative_path)**, 841
 - %option_value(option_name)**, 841
 - Top Project directives
 - customization**, 836
 - defaultBuildOptions**, 836
 - import**, 836
 - macro**, 836
 - overview, 835
 - primaryTarget**, 836
- .graph** output file type, 93
- Graphically Edit** Project Manager menu item, 68
- Green Hills GNU options
 - cpu=**, 830
- Green Hills Project Files
 - Project Manager, 38
- grep** utility
 - licensing, 47
- grep window, *see* **Search in Files Results** window
- grouping program variables, 128
- grun** utility, 540
- gs** driver option, 187
- gsize** driver option, 257
- gsize** utility, 542
- gsrec** utility
 - examples, 548 to 550
 - invoking from the driver, 244
 - options, 544
 - overview, 544
- gstack** utility, 551
- gstrip** utility, 554
- gtws** driver option, 293
- Guidelines For the Use Of The C Language In Critical Systems, Motor Industry Software Reliability Association, October 2004*, 197

Index

Guiding Declarations of Template

Functions Builder option, 235

--guiding_decls driver option, 235

gversion utility, 555

gwhat utility, 557

H

-H driver option, 101, 192

.h file extension, 41

.H file extension, 41

-h HTML compiler option, 314

header file directories

ansi, 705

C, 705

C++, 705

ecxx, 705

eecxx, 705

include/ppc, 705

scxx, 705

header files, 36, 766

 modifying C++ searches, 124

 modifying searches, 123

 printing included, 100

header stop point, 649 to 650, 653

See also precompiled header files

-headerfiledir **intex** option, 170

-headerfilename **intex** option, 170

headers, deprecated, 661

.heap program section, 460, 815

help, *see* online help

-help assembler option, 394

-help driver option, 95

-help linker option, 434

-Help linker option, 434

Help menu

 online help, obtaining, 10

Help Viewer

 navigating, 13 to 14

 opening additional manuals in, 16

 window, 11

helpview command, 10

-hex driver option, 245

HEX386 format, *see* Intel hexadecimal formats

HEX86 format, *see* Intel hexadecimal formats

.hh file extension, 41

Highlight Selections Project Manager menu item, 72

Host & Target Character Encoding

 Builder option, 229, 603, 657

HTML compiler options

-c, 268

--enableCompression, 268

-h, 314

-o, 268

-p, 268

HTML File Compression Builder

 option, 268

I

-I assembler option, 394

-I driver option, 122, 172

.i file extension, 87

.i files, 600

-I- driver option, 123

I/O

 mixing languages, 766

I/O, memory-mapped, 148

#ident, 583

.ident assembler directive, 428

-ident driver option, 309

Identifier Definition Builder option, 309

Identifier Support Builder option, 309

identifiers, 396, 769

 in assembler, 396

-identoutput driver option, 309

.idep output file type, 93

.idl

 file extension, 41

Index

- `__IeeeFloat__`, 671
- `#if`
 - precompiled header files and, 649
 - `.if` assembler directive, 416
- `%if` preprocessor directive, customization file, 863
- `%ifdef` preprocessor directive, customization file, 863
- Ignore build errors** Project Manager menu item, 75
- Ignore dependencies** Project Manager menu item, 75
- ignore_debug_references** driver option, 252
- `.ii` files, 635 to 636
- `.ii` output file type, 93
- illegal assumptions
 - implied register usage, 783
 - memory allocation, 784
- images for linking, converting customization files, 851
- Implicit Dependency Analysis (IDA)
 - gbuild** utility, 505
- Implicit Int Return Type** Builder option, 263
- Implicit Source File Inclusion** Builder option, 236, 239, 624, 640 to 641
- Implicit Use of 'std' Namespace** Builder option, 230
- implicit_extern_c_type_conversion** driver option, 308
- implicit_include** driver option, 236, 640
- implicit_typename** driver option, 236
- Import Symbols** Builder option, 254
- import** Top Project directive, 836
- `#include`
 - precompiled header files and, 648
 - `.include` assembler directive, 418
- `#include` directive, 173
- Include Directories** Builder option, 61, 122 to 124, 172
- include** driver option, 300
- Include File Search Directory** Builder option, 170
- include files
 - directive for, 418
 - directory for assembler, 394
 - list of directories searched for, 173
- Include libbsp.a** Builder option, 167
- %include** preprocessor directive, customization file, 863
- include_never** driver option, 299
- include_once** driver option, 299
- include/ppc** header file directory, 705
- includefiledir** `intex` option, 170
- Incorrect Pragma Directives** Builder option, 262, 679
- incorrect_pragma_errors** driver option, 262, 679
- incorrect_pragma_silent** driver option, 262, 679
- incorrect_pragma_warnings** driver option, 262
- ind_bcnt.c** run-time code, 758
- ind_call.ppc** run-time code, 756
- ind_crt0.c** startup code, 755
- ind_crt1.c** run-time code, 757
- ind_dots.ppc** run-time code, 756
- ind_errn.c** run-time code, 757
- ind_exit.c** run-time code, 759
- ind_gcnt.ppc** run-time code, 758
- ind_gpid.c** run-time code, 757
- ind_gprf.c** run-time code, 758
- ind_heap.c** run-time code, 758
- ind_io.c** run-time code, 759
- ind_iob.c** run-time code, 759
- ind_lock.c** run-time code, 757
- ind_manprf.c** run-time code, 757
- ind_mcnt.ppc** run-time code, 758
- ind_mcpy.ppc** startup code, 755
- ind_mprf.c** run-time code, 758
- ind_mset.ppc** startup code, 755

Index

- ind_reset.ppc** run-time code, 756
- ind_reset.ppc** startup code, 755
- ind_stackcheck.c** run-time code, 757
- ind_targ1.c** run-time code, 757
- ind_thrd.c** run-time code, 758
- .inf** output file type, 93
- %info** preprocessor directive, customization file, 863
- initialized data, in ROM, 458
- Inline C Memory Functions** Builder option, 288, 326
- Inline C String Functions** Builder option, 288, 326
- Inline Larger Functions** Builder option, 180, 324
- Inline Specific Functions** Builder option, 186, 322 to 324
- Inline Tiny Functions** Builder option, 287, 317
- inline_prologue** driver option, 273
- inline_tiny_functions** driver option, 288
- inline_trivial** driver option, 251
- inlining
 - advantages of, 325
 - automatic two-pass, 321
 - C memory functions, 325
 - C string functions, 326
 - C++, 328
 - description, 317
 - manual, 318
 - `__noinline`, 329
 - single-pass, 320
 - specific functions, 322
 - template function, 329
 - two-pass, 321
- inlining** driver option, 303, 328
- inlining optimizations, 317
- inlining small functions
 - interprocedural optimizations, 331
- inlining_unless_debug** driver option, 303, 328
- Input Languages** Builder option, 88, 275
- INPUTDIR context sensitive variable, customization file, 859
- INPUTFILE context sensitive variable, customization file, 859
- .inspect** assembler directive, 426
- Instantiate Extern Inline** Builder option, 231, 624
- instantiate_extern_inline** driver option, 231
- instantiation_dir** driver option, 237
- .int**
 - file extension, 41
- int data type, 106
 - size, 669, 677
- `__INT_BIT`, 669
- `__Int_Is_32`, 677
- `__Int_Is_64`, 677
- Integrated Development Environment (IDE), *see* MULTI Integrated Development Environment (IDE)
- `__INTEGRITY`, 676
- INTEGRITY Application Stripping** Builder option, 171
- INTEGRITY Posix Mode** Builder option, 168
- INTEGRITY Relocatable Program** Builder option, 167
- integrity_posix** driver option, 168
- :integrity_posix_mode=noposix** driver option, 168
- :integrity_posix_mode=single** driver option, 168
- :integrity_posix_mode=system** driver option, 168
- Intel hexadecimal formats
 - generating, 244, 544
- Interleaved Source and Assembly** Builder option, 242
- Intermediate Output Directory Relative to This File** Builder option, 276

Index

Intermediate Output Directory Relative to Top-Level Project Builder

option, 175

Intermodule Inlining Builder option, 178, 316, 321, 323 to 324

interprocedural optimizations, 330

alias analysis, 330

constant returns, propagation of, 331

inlining small functions, 331

read-based alias analysis, 330

using **gbuild**, 337

whole program optimizations, 333

write-based alias analysis, 330

Interprocedural Optimizations Builder

option, 179, 330, 334, 336 to 338

interrupt

functions, 147, 685

processing, 147

routine and optimizations, 590

`__interrupt` keyword, 147

-interrupt_prologue_always_uses_stm

driver option, 272

intex options

-display_boot, 169

-dynamic, 169

-generatemonolithdeletions, 169

-headerfiledir, 170

-headerfilename, 170

-includefiledir, 170

-intex_var, 171

-intexoption, 170

-memory, 169

-memory=, 170

-no_display_boot, 169

-no_dynamic, 169

--no_quit_after_intex_warnings, 171

-no_strip_application, 171

--quit_after_intex_warnings, 171

-sha1, 171

-strip_application, 171

Intex Variable Definition Builder

option, 171

-intex_var intex option, 171

-intexoption intex option, 170

intrinsic functions

`__abs()`, 695

`__builtin_return_address()`, 692

`__builtin_set_return_address()`, 693

`__CLZ32()`, 697

`__DCBF()`, 696

`__DCBST()`, 696

`__DCBT()`, 696

`__DCBZ()`, 696

`__DI()`, 693

`__DIR()`, 694

`__EI()`, 693

`EIEIO()`, 693

`__EIR()`, 694

`__fabs()`, 695

`__fabsf()`, 695

`__FMADD()`, 696

`__FMADDS()`, 696

`__FMSUB()`, 696

`__FMSUBS()`, 696

`__fnabs()`, 695

`__fnabsf()`, 695

`__FNMADD()`, 696

`__FNMADDS()`, 696

`__FNMSUB()`, 696

`__FNMSUBS()`, 696

`__FRES()`, 696

`__FRSQRT()`, 696

`__FSEL()`, 696

`__GETSR()`, 695

`__ISYNC()`, 693

`__labs()`, 695

`__lhbrx()`, 695

`__LWARX()`, 692

`__lwbrx()`, 695

`__LWSYNC()`, 693

`__MBAR()`, 693

Index

- __MFD CR(), 693
- __MFFS(), 694
- __MFSPR(), 694
- __MFTB(), 692
- __MFTBU(), 692
- __MSYNC(), 693
- __MTDCR(), 693
- __MTFSF(), 694
- __MTSPR(), 694
- __MULSH(), 697
- __MULUH(), 697
- __PS_ABS(), 698
- __PS_ADD(), 698
- __PS_ADDS(), 699
- __PS_DIV(), 698
- __PS_DIVS(), 699
- __PS_FDUP(), 699
- __PS_MADD(), 698
- __PS_MADDS(), 699
- __PS_MADDS0(), 698
- __PS_MADDS0F(), 699
- __PS_MADDS1(), 698
- __PS_MERGE00(), 698
- __PS_MERGE01(), 698
- __PS_MERGE10(), 698
- __PS_MERGE11(), 698
- __PS_MSUB(), 698
- __PS_MSUBS(), 699
- __PS_MUL(), 698
- __PS_MULS(), 699
- __PS_MULS0(), 698
- __PS_MULS0F(), 699
- __PS_MULS1(), 698
- __PS_NABS(), 698
- __PS_NEG(), 698
- __PS_NEGS(), 699
- __PS_NMADD(), 698
- __PS_NMADDS(), 699
- __PS_NMSUB(), 698
- __PS_NMSUBS(), 699
- __PS_RES(), 698

- __PS_RESS(), 699
- __PS_RSQRTE(), 698
- __PS_SEL(), 698
- __PS_SUB(), 698
- __PS_SUBS(), 699
- __PS_SUM0(), 698
- __PS_SUM0F(), 699
- __PS_SUM1(), 698
- __PS_SUM1_F(), 699
- __PS_SUM1F(), 699
- __PS_SUM1FF(), 699
- __PSQ_L(), 699
- __PSQ_LX(), 699
- __PSQ_ST(), 699
- __PSQ_STX(), 699
- __PTESYNC(), 693
- __RIR(), 694
- __RLWIMI(), 695
- __RLWINM(), 695
- __RLWNM(), 695
- __setflm(), 694
- __SETSR(), 695
- __sthbrx(), 695
- __stwbrx(), 695
- __STWCX(), 692
- __SYNC(), 693
- __TLBSYNC(), 693
- __WAIT(), 693

intrinsic s, 692

IP Alias Lib Functions Builder

option, 185, 331

IP Alias Reads Builder option, 184, 330

IP Alias Writes Builder option, 184, 331

IP Constant Globals Builder option, 182, 336

IP Constant Propagation Builder
option, 183, 335

IP Constant Returns Builder option, 184, 331

IP Delete Functions Builder option, 182, 336

Index

IP Delete Globals Builder option, 182, 336
IP Limit Inlining Builder option, 183
IP One-Site Inlining Builder option, 182, 335
IP Remove Parameters Builder option, 183, 335
IP Remove Returns Builder option, 184, 335
IP Small Inlining Builder option, 183, 331
.ipf output file type, 94
-irel driver option, 167
-irelprog driver option, 167
isalnum(), ANSI C implementation, 614
isalpha(), ANSI C implementation, 614
iscntrl(), ANSI C implementation, 614
isdefined linker expression function, 450
-isel driver option, 270
islower(), ANSI C implementation, 614
ISO/IEC 14882:1998, 624
isprint(), ANSI C implementation, 614
isupper(), ANSI C implementation, 614
isweak linker expression function, 450
__ISYNC() intrinsic function, 693
ITEMID context sensitive variable, customization file, 859

J

Japanese Automotive C, 587
builder and driver options for, 587 to 588
-japanese_automotive_c driver option, 195, 587 to 588

__Japanese_Automotive_C__, 666

K

K & R C

asm statements, 583
const keyword, 583
#defined(id), 583
#elif, 583
#error, 583
extensions, 583
extern storage class specifier, 583
#ident, 583
signed keyword, 583
struct data type, 583
union data type, 583
variable arguments, 595
volatile keyword, 583
-k+r driver option, 194, 564, 582
Kanji, 603
-kanji driver option, 229, 603
-keep linker option, 434
--keep_static_symbols driver option, 228
-keepmap driver option, 255
KEEPOBJECTS context sensitive variable, customization file, 859
-keeptempfiles driver option, 275
-kernel driver option, 166
Kernel Project Builder option, 166
-kernel=kernel_checked driver option, 166
-kernel=kernel_opt driver option, 166
-kernel=kernel_vel driver option, 166
-kernel=kernel_vel_checked driver option, 166
-kernel=kernel_vel_opt driver option, 166
keyword help, 10
Koenig lookup, *see* argument-dependent lookup

Index

L

-l driver option, 125, 173

-L driver option, 125, 173

labels

assembly, 399, 407

`__labs()` intrinsic function, 695

-language driver option, 275

`__LANGUAGE_ASM__`, 666

`__LANGUAGE_C__`, 666

`__LANGUAGE_CXX__`, 666

language-independent library, 750

Launcher, *see* MULTI Launcher

-layout=ram

locating program sections in ROM and RAM, 132

-layout=romcopy

locating program sections in ROM and RAM, 132, 755

-layout=romrun

locating program sections in ROM and RAM, 132

`.lcomm` assembler directive, 426

.lcp output file type, 94

.ld file extension, 41, 88

See also linker directives files (**.ld**)

legacy GNU options

###, 830

-EB, 830

-EL, 830

-maltivec, 830

-march=, 830

-mcpu, 830

-mlong-calls, 830

-mno-altivec, 830

-mno-long-calls, 830

-msdata=none, 830

less buffered I/O, 709

`__lhrx()` intrinsic function, 695

libansi.a library, 706

libarch.a library, 707

-libbsp driver option, 167

libdbmem.a library, 707

libece.a library, 706

libecnoe.a library, 706

libedge.a library, 706

libedgnoe.a library, 706

libeece.a library, 706

libeecnoe.a library, 706

libind.a

functions, 750

language-independent library, 750

libind.a library, 706

libmath.a

functions, 738, 745, 749

libmath.a library, 706

libmulti.a library, 707

libnoflt.a library, 706

librarian

command modifiers

c, 477

C, 477

e, 477

m, 477

M, 477

n, 477

o, 477

s, 477

S, 477

v, 477

commands

-d, 476

-p, 476

-r, 476

-t, 476

-x, 476

creating libraries with, 474

examples, 478

introduction to, 8

options

-display_toc, 478

-pecoff, 478

-stderr=errfile, 478

Index

- tmp=dir**, 478
 - overview, 474
 - syntax, 475
 - table of contents, creating and updating, 478
- libraries
 - building with your project, 36
 - creating and using, examples, 478
 - deleting files from, 476
 - EC++, 667
 - incorporating changes, 751
 - initialization, 765
 - libansi.a**, 706
 - libarch.a**, 707
 - libdbmem.a**, 707
 - libece.a**, 706
 - libecnoe.a**, 706
 - libedge.a**, 706
 - libedgnoe.a**, 706
 - libeece.a**, 706
 - libeecnoe.a**, 706
 - libind.a**, 684, 706
 - libmath.a**, 706
 - libmulti.a**, 707
 - libnoflt.a**, 706
 - libsce.a**, 706
 - libscnoe.a**, 706
 - libsedge.a**, 706
 - libsedgnoe.a**, 706
 - libstartup.a**, 707
 - libsys.a**, 707
 - linking to, 37
 - merging, 475
 - searching, 173
 - setting options for, 62
- Libraries and PrepRPL** Builder
 - option, 164
- Libraries** Builder option, 125, 173
- library directories, 705
 - ppc**, 705
- Library Directories** Builder option, 125, 173
- library function
 - alias analysis, 331
- libsce.a** library, 706
- libscnoe.a** library, 706
- libsedge.a** library, 706
- libsedgnoe.a** library, 706
- libstartup.a**
 - functions, 750
- libstartup.a** library, 459, 707, 755
- libsys.a**
 - system calls, 737
- libsys.a** library, 459, 707, 756
- License Info** Project Manager menu
 - item, 81
- limitations
 - alignment and size, 777
 - C compiler, 604
 - memory optimization, 784
- <limits.h>** header file, 776
- #line**
 - precompiled headers, and, 651
- line terminators
 - in assembler source statements, 400
- Line Wrap Messages** Builder option, 261
 - __LINE__**, 583, 665
- .linfix** program section, 813
- Link in Standard Libraries** Builder
 - option, 310
- Link Mode** Builder option, 311
- link_constprop** driver option, 251
- link_filter** driver option, 308
- Link-Once Template Instantiation**
 - Builder option, 235, 635, 638
- link_once_templates** driver option, 235, 638
- link_output_mode_linksafe** driver
 - option, 311
- link_output_mode_reuse** driver
 - option, 311

Index

- link_output_mode_safe** driver
 - option, 311
- link_output_mode_unlink** driver
 - option, 311
- Link-Time Constant Propagation Builder**
 - option, 251
- Link-Time Ignore Debug References**
 - Builder option, 252
- Link-Time Trivial Function Inlining**
 - Builder option, 251
- linkage, 654
- linker
 - advanced features, 469
 - compressed ROM, 453, 456
 - ELF optional header output, 822
 - introduction to, 8
 - invoking with the driver, 126, 433
 - invoking with the Project Manager, 126
 - memory maps, 440
 - See also* linker directives files (.ld)
 - overview, 432
 - program entry point, 437
 - SDA overflow errors, 145
 - section maps, 442
 - ZDA overflow errors, 145
- Linker Command File Builder**
 - option, 249
- Linker Directive Files with Non-standard Extensions** Builder option, 247
- Linker Directives Directory Builder**
 - option, 310
- linker directives files (.ld)
 - associating with a Project Manager
 - project, 64
 - Builder and driver options for, 248 to 249, 310
 - customizing run-time sections, 459
 - defaults
 - standalone_ram.ld**, 63
 - standalone_romcopy.ld**, 64
 - standalone_romrun.ld**, 64
 - DEFAULTS directive, 439
 - directory, specifying, 310
 - locating program sections in ROM and RAM, 132
 - maximizing speed, 458
 - MEMORY directive, 440
 - memory maps, 440
 - minimizing RAM usage, 458
 - OPTION directive, 438
 - overview, 437
 - Project Manager, specifying with, 32
 - ROM and CROM, 456
 - SDA and, 142
 - section attributes, 451
 - section maps, 442
 - SECTIONS directive, 442
 - sections, including and renaming, 445
 - sections, reserved memory area, 441
 - specifying options after, 248
 - specifying options before, 249
 - using, 88
- linker expression functions
 - absolute, 449
 - addr, 449
 - align, 449
 - alignof, 449
 - endaddr, 449
 - error, 449
 - final, 449
 - isdefined, 450
 - isweak, 450
 - max, 450
 - memaddr, 450
 - min, 450
 - pack_or_align, 450
 - provide, 450
 - sizeof, 450
- Linker generated tables
 - __ghstable_tablename**,
 - __ghsentry_tablename_entryname**, 465

Index

Linker Optimizations Builder

option, 178, 250

linker options

- allabsolute**, 434
- allow_bad_relocations**, 434
- append**, 434
- earlyabsolute**, 434
- extractall**, 434, 466
- extractweak**, 434, 466
- help**, 434
- Help**, 434
- keep**, 434
- many_segments**, 434
- no_append**, 435
- no_crom**, 435, 457
- no_uncompressed_copy**, 435
- nosegments_always_executable**, 435
- prog2**, 435
- T**, 435
- unalign_debug_sections**, 435
- underscore**, 435
- v**, 435
- V**, 436
- w**, 436
- wrapsym**, 436

linker raw files, 41

linker section attributes

- ABS, 451
- ALIGN, 451
- AT, 451
- CLEAR, 452
- CROM, 453, 456
- FILL, 452
- LOAD, 451
- MAX_ENDADDRESS, 452
- MAX_SIZE, 452
- MIN_ENDADDRESS, 452
- MIN_SIZE, 452
- NOCHECKSUM, 452
- NOCLEAR, 452
- NOLOAD, 452

PAD, 452

ROM, 453, 456

ROM_NOCOPY, 453

SHFLAGS, 453

Linker Symbols

__ghsalwaysimport, 466

__ghsautoimport, 466

Linker Warnings Builder option, 246

Linker-Based Far Call Patching Builder

option, 135, 159

--linker_link driver option, 308

-linker_warnings driver option, 246

Linking Sections with Different Types

Builder option, 311

linkonce keyword, 570

__linkonce keyword, 570

.list assembler directive, 424

-list assembler option, 395

-list driver option, 125, 241

-list_dir driver option, 241

listing format directives, 424

__LITTLE_ENDIAN__, 669

-littleendian driver option, 154, 830

__littleendian keyword, 589

__LL_BIT, 677

__LL_Is_64, 677

__LLONG_BIT, 669

-lnk driver option, 248

.lnk file extension, 41

-lnk0 driver option, 248

-lnkcmd driver option, 249

Load Configuration Project Manager

menu item, 78

LOAD linker section attribute, 451

Load Module Project Manager menu

item, 76

local scalar variables, 362

-locals_unchanged_by_longjmp driver option, 304

-locatedprogram driver option, 244

log(), ANSI C implementation, 615

Index

`logf()`, ANSI C implementation, 615

`.long` assembler directive, 417

long data type, 106

size, 669, 677

long double data type, 106

long long data type, 106, 226

size, 669, 677

Long Long Support Builder option, 226

`__LONG_BIT`, 669

`__Long_Is_32`, 677

`__Long_Is_64`, 677

--long_long driver option, 226

longjmp() Does Not Restore Local Vars

Builder option, 304

loop invariant analysis optimization, 341

loop optimizations, 339

automatic vector optimization, 344

manual vectorization, 343

vectorization, 343

Loop Optimize Specific Functions

Builder option, 186, 340

loop rotation optimization, 364

Loop Unrolling Builder option, 288, 339, 342

loop unrolling optimization, 342

`.lsbss` assembler directive, 427

.lst output file type, 94

`__LWARX()` intrinsic, 692

`__lwbrx()` intrinsic function, 695

`__LWSYNC()` intrinsic function, 693

M

-Ma driver option, 256

`__MAC__`, 668

machine-specific arithmetic, 782

macro

definition directives, 418

macro assembler, 392

`.macro` assembler directive, 420

macro functions, for **.gpj** files, 841

macro Top Project directive, 836

macros

defining, 418

`main()`

mixing languages, 765

`_main()`, 765, 767 to 768

-Make driver option, 92

`MAKE_DEPEND_OPTIONS` context

sensitive variable, customization

file, 859

makefiles

and custom run-tim environments, 752

dependency information, 100

generating, 539

incremental builds with, 98

macros for, 98

multiple languages and, 99

using, 97

-makerpl.args driver option, 165

`malloc()`

memory allocation, 714

`malloc()`, ANSI C implementation, 618

`malloc_size` DoubleCheck property

type, 371

`malloced` DoubleCheck property

type, 370

-maltivec legacy GNU option, 830

managing your project

Project Manager, 27

manual inlining, 318

Manuals Project Manager menu item, 80

-many_segments linker option, 434

-map driver option, 255

Map File Cross-Referencing Builder

option, 256

Map File Generation Builder option, 255

Map File Page Length Builder

option, 255

Map File Retention Builder option, 255

Map File Sorting Builder option, 256

.map output file type, 94

Index

- maplines** driver option, 255
- march**= legacy GNU option, 830
- math.h**
 - functions, 738
- max linker expression function, 450
- MAX_ENDADDRESS linker section attribute, 452
- max_inlining** driver option, 303, 328
- max_inlining_unless_debug** driver option, 303, 328
- MAX_SIZE linker section attribute, 452
- Maximize Optimizations** Builder option, 180
- Maximum Number of Errors to Display**
 - Builder option, 259
- __MBAR() intrinsic function, 693
- .mbs** file extension, 41
- mcpu** legacy GNU option, 830
- MD** driver option, 101
- .mem** output file type, 94
- memaddr linker expression function, 450
- memory, 104
 - alignment, 591
 - defining, 440
 - optimization
 - restrictions, 784
 - problems with size, 785
- memory allocation
 - alloca(), 711
 - malloc(), 714
- memory** driver option, 245
- memory_intex** option, 169
- MEMORY linker directive, 440
- memory maps, 440
 - See also* linker directives files (.ld)
- memory optimization, 359
- Memory Optimization** Builder
 - option, 290, 359 to 360, 590, 784 to 785
- memory= intex** option, 170
- memory-mapped I/O, 148
- merge_archive** driver option, 92, 475
- Message Pragma** Builder option, 301
 - __MFDCR() intrinsic function, 693
 - __MFFS() intrinsic function, 694
 - __MFSPR() intrinsic function, 694
 - __MFTBU() intrinsic function, 692
- min linker expression function, 450
- MIN_ENDADDRESS linker section attribute, 452
- MIN_SIZE linker section attribute, 452
- misaligned loads and stores, handling, 111
- MISRA C, 587
 - builder and driver options for, 216 to 225
 - #pragma directives, 689
 - predefined macros for, 672
- MISRA C - Advisory Rules Level** Builder option, 215
- MISRA C - Required Rules Level** Builder option, 215
- MISRA C - Run-Time Checks** Builder option, 216
- MISRA C 1998
 - builder and driver options for, 216
 - introduction to, 216
- MISRA C 1998 Rules - Character Set** Builder option, 217
- MISRA C 1998 Rules - Comments** Builder option, 217
- MISRA C 1998 Rules - Constants** Builder option, 218
- MISRA C 1998 Rules - Control Flow** Builder option, 221
- MISRA C 1998 Rules - Conversions** Builder option, 220
- MISRA C 1998 Rules - Declarations & Definitions** Builder option, 218
- MISRA C 1998 Rules - Environment** Builder option, 216
- MISRA C 1998 Rules - Expressions** Builder option, 220

Index

- MISRA C 1998 Rules - Functions Builder**
 - option, 221
- MISRA C 1998 Rules - Identifiers**
 - Builder option, 217
- MISRA C 1998 Rules - Initialization**
 - Builder option, 219
- MISRA C 1998 Rules - Operators**
 - Builder option, 219
- MISRA C 1998 Rules - Pointers and Arrays**
 - Builder option, 223
- MISRA C 1998 Rules - Preprocessor**
 - Builder option, 222
- MISRA C 1998 Rules - Standard Library**
 - Builder option, 224
- MISRA C 1998 Rules - Structs and Unions**
 - Builder option, 223
- MISRA C 1998 Rules - Types Builder**
 - option, 218
- MISRA C 2004**
 - builder and driver options for, 197
 - introduction to, 197
 - switch statements, 199
- MISRA C 2004 Rules - 1 Environment**
 - Builder option, 200
- MISRA C 2004 Rules - 10 Arithmetic Type Conversions**
 - Builder option, 205
- MISRA C 2004 Rules - 11 Pointer Type Conversions**
 - Builder option, 206
- MISRA C 2004 Rules - 12 Expressions**
 - Builder option, 206
- MISRA C 2004 Rules - 13 Control Statement Expressions**
 - Builder option, 208
- MISRA C 2004 Rules - 14 Control Flow**
 - Builder option, 209
- MISRA C 2004 Rules - 15 Switch Statements**
 - Builder option, 210
- MISRA C 2004 Rules - 16 Functions**
 - Builder option, 210
- MISRA C 2004 Rules - 17 Pointers and Arrays**
 - Builder option, 211
- MISRA C 2004 Rules - 18 Structs and Unions**
 - Builder option, 212
- MISRA C 2004 Rules - 19 Preprocessing Directives**
 - Builder option, 212
- MISRA C 2004 Rules - 2 Language Extensions**
 - Builder option, 200
- MISRA C 2004 Rules - 20 Standard Libraries**
 - Builder option, 214
- MISRA C 2004 Rules - 21 Run-time Failures**
 - Builder option, 215
- MISRA C 2004 Rules - 3 Documentation**
 - Builder option, 201
- MISRA C 2004 Rules - 4 Character Sets**
 - Builder option, 201
- MISRA C 2004 Rules - 5 Identifiers**
 - Builder option, 202
- MISRA C 2004 Rules - 6 Types**
 - Builder option, 203
- MISRA C 2004 Rules - 7 Constants**
 - Builder option, 203
- MISRA C 2004 Rules - 8 Declarations and Definitions**
 - Builder option, 203
- MISRA C 2004 Rules - 9 Initialization**
 - Builder option, 205
- misra_2004=1.1** driver option, 200
- misra_2004=1.2** driver option, 200
- misra_2004=1.3** driver option, 200
- misra_2004=1.4** driver option, 200
- misra_2004=1.5** driver option, 200
- misra_2004=10.1** driver option, 205
- misra_2004=10.2** driver option, 205
- misra_2004=10.3** driver option, 205
- misra_2004=10.4** driver option, 205
- misra_2004=10.5** driver option, 206
- misra_2004=10.6** driver option, 206
- misra_2004=11.1** driver option, 206
- misra_2004=11.2** driver option, 206
- misra_2004=11.3** driver option, 206
- misra_2004=11.4** driver option, 206
- misra_2004=11.5** driver option, 206
- misra_2004=12.1** driver option, 206

Index

- misra_2004=12.10 driver option, 207
- misra_2004=12.11 driver option, 207
- misra_2004=12.12 driver option, 208
- misra_2004=12.13 driver option, 208
- misra_2004=12.2 driver option, 206
- misra_2004=12.3 driver option, 207
- misra_2004=12.4 driver option, 207
- misra_2004=12.5 driver option, 207
- misra_2004=12.6 driver option, 207
- misra_2004=12.7 driver option, 207
- misra_2004=12.8 driver option, 207
- misra_2004=12.9 driver option, 207
- misra_2004=13.1 driver option, 208
- misra_2004=13.2 driver option, 208
- misra_2004=13.3 driver option, 208
- misra_2004=13.4 driver option, 208
- misra_2004=13.5 driver option, 208
- misra_2004=13.6 driver option, 208
- misra_2004=13.7 driver option, 208
- misra_2004=14.1 driver option, 209
- misra_2004=14.10 driver option, 209
- misra_2004=14.2 driver option, 209
- misra_2004=14.3 driver option, 209
- misra_2004=14.4 driver option, 209
- misra_2004=14.5 driver option, 209
- misra_2004=14.6 driver option, 209
- misra_2004=14.7 driver option, 209
- misra_2004=14.8 driver option, 209
- misra_2004=14.9 driver option, 209
- misra_2004=15.1 driver option, 210
- misra_2004=15.2 driver option, 210
- misra_2004=15.3 driver option, 210
- misra_2004=15.4 driver option, 210
- misra_2004=15.5 driver option, 210
- misra_2004=16.1 driver option, 210
- misra_2004=16.10 driver option, 211
- misra_2004=16.2 driver option, 210
- misra_2004=16.3 driver option, 210
- misra_2004=16.4 driver option, 211
- misra_2004=16.5 driver option, 211
- misra_2004=16.6 driver option, 211
- misra_2004=16.7 driver option, 211
- misra_2004=16.8 driver option, 211
- misra_2004=16.9 driver option, 211
- misra_2004=17.1 driver option, 211
- misra_2004=17.2 driver option, 211
- misra_2004=17.3 driver option, 211
- misra_2004=17.4 driver option, 211
- misra_2004=17.5 driver option, 212
- misra_2004=17.6 driver option, 212
- misra_2004=18.1 driver option, 212
- misra_2004=18.2 driver option, 212
- misra_2004=18.3 driver option, 212
- misra_2004=18.4 driver option, 212
- misra_2004=19.1 driver option, 212
- misra_2004=19.10 driver option, 213
- misra_2004=19.11 driver option, 213
- misra_2004=19.12 driver option, 213
- misra_2004=19.13 driver option, 213
- misra_2004=19.14 driver option, 213
- misra_2004=19.15 driver option, 214
- misra_2004=19.16 driver option, 214
- misra_2004=19.17 driver option, 214
- misra_2004=19.2 driver option, 212
- misra_2004=19.3 driver option, 212
- misra_2004=19.4 driver option, 212
- misra_2004=19.5 driver option, 212
- misra_2004=19.6 driver option, 213
- misra_2004=19.7 driver option, 213
- misra_2004=19.8 driver option, 213
- misra_2004=19.9 driver option, 213
- misra_2004=2.1 driver option, 200
- misra_2004=2.2 driver option, 201
- misra_2004=2.3 driver option, 201
- misra_2004=2.4 driver option, 201
- misra_2004=20.1 driver option, 214
- misra_2004=20.10 driver option, 215
- misra_2004=20.11 driver option, 215
- misra_2004=20.12 driver option, 215
- misra_2004=20.2 driver option, 214
- misra_2004=20.3 driver option, 214
- misra_2004=20.4 driver option, 214

Index

- misra_2004=20.5 driver option, 214
- misra_2004=20.6 driver option, 214
- misra_2004=20.7 driver option, 215
- misra_2004=20.8 driver option, 215
- misra_2004=20.9 driver option, 215
- misra_2004=21.1 driver option, 215
- misra_2004=3.1 driver option, 201
- misra_2004=3.2 driver option, 201
- misra_2004=3.3 driver option, 201
- misra_2004=3.4 driver option, 201
- misra_2004=3.5 driver option, 201
- misra_2004=3.6 driver option, 201
- misra_2004=4.1 driver option, 201
- misra_2004=4.2 driver option, 202
- misra_2004=5.1 driver option, 202
- misra_2004=5.2 driver option, 202
- misra_2004=5.3 driver option, 202
- misra_2004=5.4 driver option, 202
- misra_2004=5.5 driver option, 202
- misra_2004=5.6 driver option, 202
- misra_2004=5.7 driver option, 202
- misra_2004=6.1 driver option, 203
- misra_2004=6.2 driver option, 203
- misra_2004=6.3 driver option, 203
- misra_2004=6.4 driver option, 203
- misra_2004=6.5 driver option, 203
- misra_2004=7.1 driver option, 203
- misra_2004=8.1 driver option, 203
- misra_2004=8.10 driver option, 204
- misra_2004=8.11 driver option, 204
- misra_2004=8.12 driver option, 205
- misra_2004=8.2 driver option, 203
- misra_2004=8.3 driver option, 204
- misra_2004=8.4 driver option, 204
- misra_2004=8.5 driver option, 204
- misra_2004=8.6 driver option, 204
- misra_2004=8.7 driver option, 204
- misra_2004=8.8 driver option, 204
- misra_2004=8.9 driver option, 204
- misra_2004=9.1 driver option, 205
- misra_2004=9.2 driver option, 205
- misra_2004=9.3 driver option, 205
- misra_adv=error driver option, 216
- misra_adv=silent driver option, 216
- misra_adv=warn driver option, 216
- __MISRA_i, 672
- misra_req=error driver option, 215
- misra_req=silent driver option, 215
- misra_req=warn driver option, 215
- misra_runtime driver option, 216
- mixed language executable, 764
- MI driver option, 256
- mlong-calls legacy GNU option, 830
- MMD driver option, 101
- Mn driver option, 256
- mno-altivec legacy GNU option, 830
- mno-long-calls legacy GNU option, 830
- Modify Project** Project Manager menu item, 68
- .mon output file type, 93
- Motor Industry Software Reliability Association C Rules, *see* MISRA C
- msdata=none legacy GNU option, 830
- __msw, 677
- __MSYNC() intrinsic function, 693
- __MTDCR() intrinsic function, 693
- __MTFSF() intrinsic function, 694
- __MTSPR() intrinsic function, 694
- Mu driver option, 256
- __MULSH() intrinsic function, 697
- MULTI Integrated Development Environment (IDE)
 - document set, xxi
 - online help, 10
 - overview, 2
- MULTI Launcher
 - overview, 4
- MULTI Project Project Manager Help**
 - Project Manager menu item, 80
- MULTI Version** Builder option, 294
- multi-byte characters, 601
- multi_version driver option, 294

Index

-multiple driver option, 253

multiple-section programs, 128

Multiply-Defined Symbols Builder

option, 253

`__MULUH()` intrinsic function, 697

--munch driver option, 308

-Mx driver option, 256

N

name lookup

template instantiation, 645

name mangling, 654, 656

named labels, 407

namespace support

argument dependent lookup, 647

dependent name lookup, 645

lookup using the reference context, 645

Namespace Support Builder option, 230, 645

namespaces, 645

predefined macro for, 676

`std`, 675

template instantiation, and, 645

`__NAMESPACES`, 676

--namespaces driver option, 230

naming conventions, 769

NaN Compares as Unordered Builder

option, 156

near and far function calls, 135

near function calls, 135

`__nearcall`, 136

`.need` assembler directive, 429

`needs_null_check` DoubleCheck

property type, 370

`new`

array, predefined macro, 675

New Top Project Project Manager menu

item, 66

New Window Project Manager menu

item, 67

--new_inside_of_constructor driver option, 233

--new_outside_of_constructor driver option, 233

New-Style Cast Support Builder

option, 232

--new_style_casts driver option, 232

nm utility, 655

--no_additional_output driver option, 245

--no_alternative_tokens driver option, 229

--no_ansi_alias driver option, 197

--no_any_output_suffix driver option, 275

--no_append linker option, 435

--no_assembler_warnings driver option, 309

--no_auto_instantiation driver option, 239

--no_auto_sda driver option, 158

--no_bool driver option, 230

--no_brief_diagnostics driver option, 261

--no_c_and_cpp_functions_are_distinct driver option, 308

--no_callgraph driver option, 257

--no_codefactor driver option, 251

--no_comments driver option, 301

--no_commons driver option, 127, 228

--no_coverage_analysis driver option, 188

--no_create_rpl driver option, 163

--no_crom linker option, 435, 457

--no_data_delete driver option, 250

--no_debug driver option, 187

--no_defer_parse_function_templates driver option, 240

--no_delete driver option, 250

--no_dep_name driver option, 238

--no_discard_zero_initializers driver option, 159

Index

- no_display_boot** `intex` option, 169
- no_display_error_number** driver option, 265
- no_dual_debug** driver option, 298
- no_dwarf2dbo** driver option, 298
- no_dynamic** `intex` option, 169
- no_e500_inst_fix** driver option, 270
- no_e500_inst_fix2** driver option, 271
- no_eieio_loads** driver option, 161
- no_eieio_stores** driver option, 161
- no_error_basename** driver option, 261
- no_event_logging** driver option, 166
- no_exceptions** driver option, 196
- no_export** driver option, 238
- no_extend_liveness** driver option, 294
- no_float_scanf** driver option, 310
- no_full_breakdots** driver option, 294
- no_full_debug_info** driver option, 298
- no_gnu_asm** driver option, 305
- no_gsize** driver option, 257
- no_guiding_decls** driver option, 235
- no_ignore_debug_references** driver option, 252
- no_implicit_extern_c_type_conversion** driver option, 308
- no_implicit_include** driver option, 236, 641
- no_implicit_typename** driver option, 236
- no_include_once** driver option, 299
- no_inline_prologue** driver option, 273
- no_inline_tiny_functions** driver option, 288, 317
- no_inline_trivial** driver option, 251
- no_inlining** driver option, 303, 328
- no_instantiate_extern_inline** driver option, 232
- no_integrity_posix** driver option, 168
- no_interrupt_prologue_always_uses_stm** driver option, 272
- no_irel** driver option, 167
- no_irelprog** driver option, 167
- no_isel** driver option, 270
- no_japanese_automotive_c** driver option, 195
- no_keep_static_symbols** driver option, 228
- no_kernel** driver option, 166
- no_libbsp** driver option, 167
- no_link_constprop** driver option, 251
- no_link_filter** driver option, 308
- no_link_once_templates** driver option, 235, 635
- no_linker_warnings** driver option, 246
- no_list** driver option, 241
- no_locals_unchanged_by_longjmp** driver option, 304
- no_long_long** driver option, 226
- no_misra_runtime** driver option, 216
- no_multiple** driver option, 253
- no_namespaces** driver option, 230, 645
- no_new_style_casts** driver option, 232
- no_old_specializations** driver option, 236
- no_one_instantiation_per_object** driver option, 237
- no_only_explicit_reg_use** driver option, 272
- no_parse_templates** driver option, 238
- no_pch_match_directory** driver option, 649
- no_pch_messages** driver option, 652
- no_posix_dll** driver option, 168
- no_precise_signed_zero** driver option, 157
- no_prelink_objects** driver option, 238
- no_preprocess_assembly_files** driver option, 241
- no_preprocess_linker_directive** driver option, 246

Index

- no_preprocess_special_assembly_files**
 - driver option, 242
- no_preprpl_all_libs** driver option, 164
- no_quit_after_intex_warnings** **intex**
 - option, 171
- no_quit_after_warnings** driver
 - option, 260
- no_readonly_typeinfo** driver
 - option, 233
- no_readonly_virtual_tables** driver
 - option, 234
- no_remarks** driver option, 259
- no_restrict** driver option, 231
- no_return** DoubleCheck property
 - type, 372
- no_return_when_non_zero**
 - DoubleCheck property type, 372
- no_return_when_zero** DoubleCheck
 - property type, 371
- no_rpl_use_dbg_source_root** driver
 - option, 164
- no_rtti** driver option, 233
- no_scan_source** driver option, 296
- no_search_for_dba** driver option, 295
- no_search_source_directory** driver
 - option, 303
- no_short_enum** driver option, 226, 588
- no_slash_comment** driver option, 196
- no_stabs2dbo** driver option, 299
- no_stack_check** driver option, 160
- no_strip_application** **intex** option, 171
- no_timer_profile** driver option, 188
- no_trace_includes** driver option, 193
- no_uncompressed_copy** linker
 - option, 435
- no_undefined** driver option, 253
- no_unique_strings** driver option, 228
- no_unsafe_predefines** driver
 - option, 300
- no_use_fsel** driver option, 270
- no_using_std** driver option, 230
- no_uvfd** driver option, 251
- no_version** driver option, 260
- no_wrap_diagnostics** driver option, 261
- noAltVec** driver option, 830
- noasm3g** driver option, 243
- noautoregister** driver option, 289, 363
- nobigswitch** driver option, 273
- nobuild .gpj** conditional control
 - statement, 839
- nochecksum** driver option, 258
- NOCHECKSUM** linker section attribute, 452
- NOCLEAR**
 - linker section attribute, 452
- noconsistentcode** driver option, 293
- nocpp** driver option, 308
- nodelay_archives** driver option, 292
- nodelay_second_pass** driver option, 292
- :nodepends** Builder option, 278
- noentry** driver option, 245
- nofarcallpatch** driver option, 135, 159
- nofarcalls** driver option, 159, 830
 - __NoFloat__**, 672
- nofloatio** driver option, 157, 706
- nofloatsingle** driver option, 156
- noga** driver option, 296
 - .nogen** assembler directive, 424
- nogen** assembler option, 395
- noglimits** driver option, 293
- nogtws** driver option, 293
- noidentoutput** driver option, 309
 - __noinline**, 329
- nokeepmap** driver option, 255
- nokeeptempfiles** driver option, 275
 - .nolist** assembler directive, 425
- NOLOAD** linker section attribute, 452
- nomap** driver option, 255
- non-local static objects, 655
- non-portable features, 774
- non-SDA variable, 773
- non_shared** driver option, 162

Index

Non-Standard Output Suffix Builder

- option, 275
- non-volatile registers, 105
- noobj** driver option, 274
- nooverlap** driver option, 257
- nooverload** driver option, 289, 362
- nopackage_infs** driver option, 292
- nopasssource** driver option, 242
- nosegments_always_executable** linker option, 435
- :noSelfdepend** Builder option, 277
- noSPE** assembler option, 394
- noSPE** driver option, 271
- nostartfiles** driver option, 252
- nostddef** driver option, 300
- nostdinc** driver option, 302
- nostdlib** driver option, 310, 706
- nostrip** driver option, 245
- .note** assembler directive, 421
- nothreshold** driver option, 158
- .novle** assembler directive, 429
- novle** driver option, 162
- .nowarning** assembler directive, 429
- NULL pointer
 - size of, 780
- numeric constants, 397

O

- o** assembler option, 395
- o** driver option, 91, 147, 174
- O** driver option, 176
- .o** file extension, 41, 88
- o** HTML compiler option, 268
- O1** driver option, 291
- O2** driver option, 291
- O3** driver option, 292
- OB** driver option, 180, 324
- obj** driver option, 274
- Object File Output Directory** Builder option, 172, 237

object files

- ELF format, 802, 815
- object_dir** driver option, 172
- OBJECTBASE context sensitive variable, customization file, 860
- OBJECTS context sensitive variable, customization file, 859
- OBJECTSUFFIX context sensitive variable, customization file, 860
- obtaining profiling information
 - debugging information, 117
- Oconstprop** driver option, 290, 358
- Ocse** driver option, 289, 355
- Odebug** driver option, 176, 351
- .offset** assembler directive, 417
- Ogeneral** driver option, 176, 316 to 317, 351
- OI** driver option, 178, 186, 316, 321 to 322
- Ointerproc** driver option, 179, 330, 339
- Oip_analysis_only** driver option, 179
- Oipaliaslibfuncs** driver option, 185
- Oipaliasreads** driver option, 184
- Oipaliaswrites** driver option, 184
- Oipconstantreturns** driver option, 184
- Oipconstglobals** driver option, 183
- Oipconstprop** driver option, 183
- Oipdeletefunctions** driver option, 182
- Oipdeleteglobals** driver option, 182
- Oiplimitinlining** driver option, 184
- Oiponesiteinlining** driver option, 182, 335
- Oipremoveparams** driver option, 183
- Oipremovereturns** driver option, 184
- Oipsmallinlining** driver option, 183
- OL** driver option, 186, 291, 339 to 340, 785
- old_specializations** driver option, 236
- Olimit** driver option, 351 to 352
- Olimit= Without Debug Information** Builder option, 293

Index

- Olimit=peephole** driver option, 180
- Olimit=pipeline** driver option, 180
- Olink** driver option, 179
- OM** driver option, 290, 359, 590, 784 to 785
- Omax** driver option, 180
- Omaxdebug** driver option, 176
- Omemfuncs** driver option, 288
- Ominmax** driver option, 290, 359
- Omoredebug** driver option, 176, 351
- one instantiation per object, 637
- One Instantiation Per Object** Builder option, 236 to 237, 637
- one_instantiation_per_object** driver option, 237, 637
- online help, 10
 - See also* Help Viewer
 - for commands, 10
 - for configuration options, 10
 - context-sensitive, 10
 - for keywords, 10
 - limitations, 14
 - online manuals, 10
 - overview, 10
- Only explicit floating point and SIMD register use** Builder option, 272
- Only remove intermediate output** Project Manager menu item, 75
- only_explicit_reg_use** driver option, 272
- __ONLY_STANDARD_KEYWORDS_IN_C**, 672
- Onobig** driver option, 180
- Onoconstprop** driver option, 290, 358
- Onocse** driver option, 289, 355
- Onoinline** driver option, 178, 321
- Onoipa** driver option, 179, 337
- Onoipaliaslibfuncs** driver option, 185, 331
- Onoipaliasreads** driver option, 184, 330
- Onoipaliaswrites** driver option, 185, 331
- Onoipconstantreturns** driver option, 184, 331
- Onoipconstglobals** driver option, 183, 336
- Onoipconstprop** driver option, 183, 335
- Onoipdeletefunctions** driver option, 182, 336
- Onoipdeleteglobals** driver option, 182, 336
- Onoiplimitinlining** driver option, 184
- Onoiponesiteinlining** driver option, 182, 335
- Onoipremoveparams** driver option, 183, 335
- Onoipremovereturns** driver option, 184, 335
- Onoipsmallinlining** driver option, 183, 331
- Onolink** driver option, 179
- Onoloop** driver option, 291, 339
- Onomax** driver option, 180
- Onomemfuncs** driver option, 288, 326
- Onomemory** driver option, 290, 359, 785
- Onominmax** driver option, 290, 359
- Onone** driver option, 177, 317
- Onopeephole** driver option, 291, 351
- Onopipeline** driver option, 291, 352
- Onoprintfuncs** driver option, 288
- Onostrfuncs** driver option, 288, 326
- Onotailrecursion** driver option, 290, 356
- Onounroll** driver option, 289, 339, 342
- Onounrollbig** driver option, 180
- Onovector** driver option, 178
- Onovectorpeel** driver option, 292
- Opeep** driver option, 291, 351
- Open Project** Project Manager menu item, 66
- Open Reference BSP Project** Project Manager menu item, 66
- operating system dependencies, 781
- operator
 - delete[], 675

Index

- `new []`, 675
- operator field, 399
- operators
 - integral expression, 401
 - precedence, 402
 - type combinations and, 406
- Opipeline** driver option, 291, 352
- Oprintfuncs** driver option, 288
- optgroup=option_group .gpj** conditional control statement, 839
- Optimization Strategy** Builder option, 50, 142, 176, 181, 185, 231, 272 to 273, 287, 316, 326, 329, 351 to 352, 355 to 356, 358 to 362, 590, 690, 785
- optimizations
 - automatic inlining, 317
 - automatic register allocation, 363
 - automatic two-pass inlining, 321
 - automatic vector optimization, 344
 - Builder and driver options for, 177 to 178, 180 to 181, 185 to 186
 - C/C++ minimum/maximum, 359
 - code factoring, 469
 - common subexpression elimination, 355
 - constant folding, 364
 - constant propagation, 358
 - dead code elimination, 360
 - default, 362
 - general, 316, 351
 - increase speed, 316
 - individual, 316
 - inlining, 317
 - inlining advantages, 325
 - inlining C memory functions, 325
 - inlining C string functions, 326
 - inlining specific functions, 322
 - loop, 339
 - loop invariant analysis, 341
 - loop rotation, 364
 - loop unrolling, 342
 - manual inlining, 318
 - manual vectorization, 343
 - memory, 359
 - partial redundancy elimination
 - optimization, 355
 - peephole, 351
 - pipeline instruction scheduling, 352
 - portability and, 783
 - reduce size, 316
 - register allocation by coloring, 362
 - register coalescing, 363
 - single-pass inlining, 320
 - small data area, 137
 - static address elimination, 361
 - strength reduction, 340
 - tail calls, 356
 - two-pass inlining, 321
 - varying inlining thresholds, 324
 - zero data area, 138
- Optimize All Appropriate Loops** Builder option, 291, 339, 344, 785
- Optimize Specific Functions for Size** Builder option, 185
- Optimize Specific Functions for Speed** Builder option, 185
- optimizing compilers, 7
- option** driver option, 313
- OPTION** linker directive, 438
- %option_value(option_name) .gpj** macro function, 841
- optional .gpj** conditional control statement, 840
- OPTIONS** context sensitive variable, customization file, 860
- Options** Project Manager menu item, 79
- Options to Apply to Project** Builder option, 286
- :optionsFile** driver option, 286
- order of evaluation, 782
- .org** assembler directive, 421

Index

OS Directory Builder option, 163
-os_dir driver option, 163
-Osize driver option, 177, 273, 316 to 317, 329, 351
-Ospace driver option, *see* **-Osize** driver option
-Ospeed driver option, 177, 316 to 317, 351
-Ostrfuncs driver option, 288
-Otailrecursion driver option, 290, 356
-Ounroll driver option, 289
-Ounrollbig driver option, 180, 342
Output File Size Analysis Builder option, 257
Output File Type Builder option, 244
output file types, 93
 .ael, 93
 .bmon.out, 93
 .d, 93, 101
 .dba, 93
 .dbo, 93
 .dep, 93, 101
 .dla, 93
 .dlo, 93
 .dnm, 93
 .gmon.out, 93
 .graph, 93
 .idep, 93
 .ii, 93
 .inf, 93
 .ipf, 94
 .lcp, 94
 .lst, 94
 .map, 94
 .mem, 94
 .mon.out, 93
 .run, 94
 .ti, 93
 .time, 94
Output Filename Builder option, 147, 174

Output Filename for Generic Types
 Builder option, 278
:outputDir Builder option, 175
OUTPUTDIR context sensitive variable, customization file, 860
:outputDirRelative Builder option, 276
OUTPUTFILE context sensitive variable, customization file, 860
:outputName Builder option, 278
OUTPUTNAMEBASE context sensitive variable, customization file, 860
-OV driver option, 178, 344
-Ovectorpeel driver option, 292, 350
-overlap driver option, 257
-overlap_warn driver option, 257
-overload driver option, 289
-Owholeprogram driver option, 179, 334, 336, 339

P

-p driver option, 117, 187
-P driver option, 92, 238, 600
-p HTML compiler option, 268
-p librarian command, 476
p_align ELF program header field, 823
p_filesz ELF program header field, 822
p_flags ELF program header field, 823 to 824
p_memsz ELF program header field, 822
p_offset ELF program header field, 822
p_paddr ELF program header field, 822
p_type ELF program header field, 822
p_vaddr ELF program header field, 822
-pack driver option, 227
-pack=*n*, 674
pack_or_align linker expression function, 450

Index

Package Analysis Files Builder

option, 292

-package_infs driver option, 292

__packed keyword, 108, 111, 589

packing, 108

Packing (Maximum Structure

Alignment) Builder option, 110 to 111, 226, 689

PAD linker section attribute, 452

-paddr_offset driver option, 312

paired single extensions, 697

parallel build mode, setting, 47

parameter registers, 105

PARENTID context sensitive variable, customization file, 860

Parse Templates in Generic Form Builder

option, 238 to 239

--parse_templates driver option, 238

partial redundancy elimination

optimization, 355

Pass Through Arguments Builder

option, 284

pass= .gpj conditional control statement, 840

-passsource driver option, 242

:passThrough Builder option, 284

Paste selected file as Link Project Manager menu item, 69

Paste selected file Local Project Manager menu item, 69

paths, formatting, 38

Pattern of Files to Pull Into Project

Builder option, 285

PCC (Portable C Compiler), 582

--pch driver option, 649, 652

--pch_dir driver option, 648, 652

.pdf file extension, 41

-pecoff librarian option, 478

peephole optimization, 351

Peephole Optimization Builder

option, 291, 351

permanent registers

in interrupt processing, 147

-pg driver option, 117, 187

Physical Address Offset From Virtual Address Builder option, 311

Pipeline & Peephole Scope Builder option, 180, 351 to 352

Pipeline Instruction Scheduling Builder option, 291, 352

pipeline instruction scheduling optimization, 352

pixel, 668

Place 'EIEIO' Around Volatile Loads Builder option, 161

Place 'EIEIO' Around Volatile Stores Builder option, 161

Placement of Class Constructor Call to New Builder option, 233

Placement of Zero-Initialized Data Builder option, 159

-pnone driver option, 187

pointer

information, 670

pointer arithmetic, 780

pointer type

size, 677

POSIX DLL Builder option, 167

--posix_dll driver option, 168

post processing, 655

:postexec Builder option, 282

:postexecSafe Builder option, 282

:postexecShell Builder option, 283

:postexecShellSafe Builder option, 283

pow(), ANSI C implementation, 615

PowerPC

EABI, 104, 107, 769

__PowerPC__, 668

powf(), ANSI C implementation, 615

.ppc file extension, 40

Index

ppc library directory, 705
__ppc*, 114
__ppc__, 668
.PPC.EMB.sbss0 program
 section, 128, 138, 815
.PPC.EMB.sdata0 program
 section, 128, 138, 459
_Pragma, 569, 690
#pragma __printf_args, 681
#pragma __scanf_args, 681
#pragma alignfunc (*n*), 680
#pragma alignref (*n*), 344
#pragma alignvar (*n*), 344, 680
#pragma asm, 680
#pragma can_instantiate
 fn, 638, 680
#pragma directives
 function call, 136
 #pragma ghs section, 129
#pragma do_not_instantiate
 fn, 638, 680
#pragma endasm, 680
#pragma ghs alias *newsym*
 oldsym, 684
#pragma ghs
 callmode=default, 136
#pragma ghs callmode=far, 136
#pragma ghs callmode=near, 136
#pragma ghs
 check=(*check-list*), 684
#pragma ghs
 end_externally_visible, 688
#pragma ghs enddata-type, 688
#pragma ghs endnoinline, 328,
 688
#pragma ghs endnomisra, 689
#pragma ghs endnowarning, 686
#pragma ghs endsda, 139
#pragma ghs endzda, 139
#pragma ghs far, 136
#pragma ghs includeonce, 685
#pragma ghs
 inlineprologue, 685
#pragma ghs interrupt, 685
#pragma ghs near, 136
#pragma ghs nofloat
 interrupt, 685
#pragma ghs
 noinlineprologue, 685
#pragma ghs nowarning, 686
#pragma ghs optasm, 687
#pragma ghs Ostring, 686
#pragma ghs reference *sym*, 686
#pragma ghs revertoptions, 687
#pragma ghs safeasm, 687
#pragma ghs section, 687
#pragma ghs
 start_externally_visible, 688
#pragma ghs startdata, 688
#pragma ghs startdata-type, 688
#pragma ghs
 startnoinline, 328,
 688
#pragma ghs startnomisra, 689
#pragma ghs startsda, 139, 688
#pragma ghs startzda, 139, 688
#pragma ghs static_call
 routine, 689
#pragma ghs
 struct_min_alignment(), 689
#pragma ghs
 struct_min_alignment(*n*), 689
#pragma ghs vectorize[=*n*], 343
#pragma ghs ZO, 686
#pragma hdrstop, 649, 653
#pragma ident ``string'', 680
#pragma inline *function-list*, 680
#pragma instantiate *fn*, 638, 680
#pragma intvect *intfunc*
 integer_constant, 680

Index

- #pragma message ("string"), 681
- #pragma message "string">, 681
- #pragma no_pch, 651, 653
- #pragma once, 681
- #pragma pack (), 681
- #pragma pack (n), 108, 110, 226, 681
- #pragma pack (pop {, name} {, n}), 681
- #pragma pack (push {, name} {, n}), 681
- #pragma unknown_control_flow (function-list), 681
- #pragma weak foo, 682
- #pragma weak foo = bar, 682
- pragma_message** driver option, 301
- pragma_message_silent** driver option, 301
- pragma_message_unknown** driver option, 301
- precise_signed_zero** driver option, 157
- precompiled header files, 648
 - automatic processing of, 649
 - builder and driver options for, 648 to 649, 652
 - file contents, 651
 - manual processing, 648
 - performance issues, 653
 - #pragma directives, 651, 653
 - requirements, 650
- predefined macro names
 - __ALTIVEC__, 668
 - __ARRAY_OPERATORS, 675
 - __BASE__, 674
 - __BIG_ENDIAN__, 669
 - __BOOL, 675
 - __c_plusplus, 666
 - __CHAR_BIT, 669
 - __Char_Is_Signed__, 670, 677
 - __Char_Is_Unsigned, 677
 - __Char_Is_Unsigned__, 670
 - __CHAR_UNSIGNED__, 670
 - __COFF__, 675
 - __cplusplus, 665
 - __DATE__, 583, 651, 665
 - __DOUBLE_HL__, 671
 - __EDG__, 667
 - __EDG_IMPLICIT_USING_STD, 675
 - __EDG_RUNTIME_USES_NAMESPACES, 675
 - __EFP_APU__, 668
 - __ELF__, 675
 - __EMBEDDED_CXX, 666
 - __EMBEDDED_CXX_HEADERS, 667
 - __Enum_Field_Is_Signed__, 670
 - __Enum_Field_Is_Unsigned__, 670
 - __EXCEPTION_HANDLING, 676
 - __EXCEPTIONS, 676
 - __EXTENDED_EMBEDDED_CXX, 667
 - __EXTENDED_EMBEDDED_CXX_HEADERS, 667
 - __Field_Is_Signed__, 670
 - __Field_Is_Unsigned__, 670
 - __FILE__, 583, 665
 - __FULL_DIR__, 674
 - __FULL_FILE__, 674
 - __FUNCPTR_BIT, 669
 - __FUNCTION__, 674
 - __ghs__, 667
 - __ghs_alignment, 674
 - __ghs_asm, 667
 - __ghs_c_int__, 667
 - __ghs_fprecise__, 672
 - __GHS_Inline_Memory_Functions, 675
 - __GHS_Inline_String_Functions, 675
 - __ghs_max_pack_value, 674
 - __GHS_NOCOMMONS__, 673
 - __GHS_Optimize_Inline, 675
 - __ghs_packing, 674
 - __GHS_REVISION_DATE, 667
 - __GHS_REVISION_VALUE, 667
 - __ghs_sda, 673

Index

- __ghs_sda_threshold, 673
- __GHS_VERSION_NUMBER, 667
- __ghs_zda, 673
- __ghs_zda_threshold, 673
- __GlobalRegisters, 673
- __gnu_asm, 668
- __GNUC__, 666
- __IeeeFloat__, 671
- __INT_BIT, 669
- __Int_Is_32, 677
- __Int_Is_64, 677
- __INTEGRITY, 676
- introduction to, 664
- __Japanese_Automotive_C__, 666
- __LANGUAGE_ASM__, 666
- __LANGUAGE_C__, 666
- __LANGUAGE_CXX__, 666
- __LINE__, 583, 665
- __LITTLE_ENDIAN__, 669
- __LL_BIT, 677
- __LL_Is_64, 677
- __LLONG_BIT, 669
- __LONG_BIT, 669
- __Long_Is_32, 677
- __Long_Is_64, 677
- __MAC__, 668
- __MISRA_i, 672
- __msw, 677
- __NAMESPACES, 676
- __NoFloat__, 672
- __ONLY_STANDARD_KEYWORDS_IN_C, 672
- pixel, 668
- __PowerPC__, 668
- __ppc*, 114
- __ppc__, 668
- __PRETTY_FUNCTION__, 674
- __PROTOTYPES__, 666
- __PTR_BIT, 670
- __Ptr_Is_32, 677
- __Ptr_Is_64, 677
- __Ptr_Is_Signed__, 670
- __Ptr_Is_Unsigned__, 670
- __REG_BIT, 670
- __Reg_Is_32, 677
- __Reg_Is_64, 677
- __RTTI, 676
- __SHRT_BIT, 670
- __SIGNED_CHARS__, 670
- __SoftwareDouble__, 672
- __SoftwareFloat__, 672
- __SPE__, 668
- __STANDARD_CXX, 667
- __STANDARD_CXX_HEADERS, 667
- __STDC__, 665
- __STDC_HOSTED__, 665
- __STDC_VERSION__, 665
- __STRICT_ANSI__, 666
- __THREADX, 676
- __TIME__, 583, 651, 665
- TX_ENABLE_EVENT_LOGGING, 676
- __unix_asm, 668
- __VEC__, 668
- vector, 669
- __vle__, 669
- __WCHAR_BIT, 670
- __WChar_Is_Int__, 677
- __WChar_Is_Long__, 678
- __WChar_Is_Short__, 678
- __WChar_Is_Signed__, 670
- __WChar_Is_Unsigned__, 670
- __WCHAR_T, 676
- :preexec** Builder option, 279
- :preexecSafe** Builder option, 280
- :preexecShell** Builder option, 280
- :preexecShellSafe** Builder option, 281
- Prelink File to Create Template Instances**
Builder option, 239
- Prelink with Instantiations** Builder
option, 237
- prelink_against** driver option, 239

Index

- prelink_objects** driver option, 238
- prelinker instantiation
 - templates, 635
- Prepend String to Every Section Name**
 - Builder option, 304
- Preprocess Assembly Files** Builder
 - option, 241, 392
- Preprocess Linker Directives Files**
 - Builder option, 246
- Preprocess *selected file*** Project Manager
 - menu item, 73
- Preprocess Special Assembly Files**
 - Builder option, 242
- preprocess_assembly_files** driver
 - option, 241, 392
- preprocess_linker_directive** driver
 - option, 246
- preprocess_linker_directive_full** driver
 - option, 246
- preprocess_special_assembly_files**
 - driver option, 242
- preprocessor
 - Builder and driver options for, 192 to 193
 - overview, 600
 - saving output from, 600
- PrepRPL Export Object File** Builder
 - option, 164
- preprpl_all_libs** driver option, 164
- preprpl.args** driver option, 165
- preprpl_output** driver option, 164
- __PRETTY_FUNCTION__**, 674
- .previous** assembler directive, 421
- primaryTarget** Top Project directive, 836
- Print Current View** Project Manager
 - menu item, 66
- Print Expanded Project** Project Manager
 - menu item, 66
- printf & scanf Argument Type Checking**
 - Builder option, 263
- printf function, 710
 - C++ I/O streams, buffering, 710
- printf() function
 - less buffered I/O, 709
- Process #include Directives** Builder
 - option, 299
- processing input files, using scripts
 - customization files, 849
- processor supported, 114
- profiling
 - Builder and driver options for, 187 to 188
 - call count, 187
 - coverage, 188
 - timing, 188
- Profiling - Call Count** Builder
 - option, 117, 187
- Profiling - Coverage** Builder option, 119, 188
- Profiling - Target-Based Timing** Builder
 - option, 121, 188
- prog2** linker option, 435
- program entry point, 437
- program headers, 822
- program sections, 138
 - begin, end, and size symbols for, 463
 - .bss, 127, 813
 - .data, 127, 458, 813
 - ELF indexes, 810
 - ELF program types, 823
 - ELF section attribute flags, 811
 - ELF section types, 811
 - .fixaddr, 459, 814
 - .fixtype, 459, 814
 - header conditions, 808
 - .heap, 460, 815
 - .linfix, 813
 - names, 812
 - .PPC.EMB.sbss0, 128, 815
 - .PPC.EMB.sdata0, 128, 459

Index

- .rel, 813
- .rela, 813
- .rodata, 127, 458, 461, 813
- .ROM, 461
- .sbss, 128, 815
- .sdatabase, 461, 815
- .sdata, 128, 815
- .sdata2, 128, 459
- .secinfo, 462, 814
- .shstrtab, 813
- .stack, 463, 815
- .strtab, 814
- .symtab, 814
- .syscall, 463, 756, 814
- .text, 127, 458, 814
- .zdata, 815
- program variables, grouping, 128
- project files
 - compatibility across hosts, 38
 - editing, 35
 - inheriting options in, 62
 - moving items, 35
 - searching for, 46
 - searching in, 46
 - types of, 39
- Project Manager
 - adding executables and examples, 26
 - Advanced Build** window, 74
 - automatically including files, 41
 - Build** menu, 73
 - Build Options** window, 50, 58
 - Build Progress** window, 42
 - building projects, 42
 - configuring existing items, 33
 - Connect** menu, 76
 - controlling the assembler with, 125
 - controlling the linker with, 126
 - creating a library, 474
 - creating a new project, 22
 - Debug** menu, 77
 - Edit** menu, 67
 - File** menu, 66
 - File Shortcut Bar**, 43
 - file types, 39
 - Green Hills Project Files, 38
 - Help** menu, 80
 - libraries, 36
 - linker directives files, working with, 63
 - main window, 20
 - managing your project, 27
 - Options** window, 51 to 56, 59
 - searching for options in, 59
 - setting options in, 49
 - settings for stand-alone programs, 31
 - settings for your project, 31
 - starting, 20, 22
 - target resources project, 28
 - Target Selector** dialog box, 71
 - Tools** menu, 78
 - Top Project, 28
 - Utility Program Launcher** dialog box, 79
 - View Menu**, 72
 - Windows** menu, 80
- Project Manager menu item
 - Create Auto-Include Subproject**, 70
- Project Manager menu items
 - About MULTI Project Manager**, 80
 - Add File after selected file**, 68
 - Add File After selected file**, 70
 - Add File into selected file**, 68
 - Add Item into selected file**, 68
 - Advanced**, 70
 - Advanced Build**, 74
 - Bookmarks**, 80
 - Build Ignoring Errors selected file**, 73
 - Build selected file**, 73
 - Checkout Browser**, 78
 - Clean all output before building**, 75
 - Clean selected file**, 74

Index

- Clear User Default Configuration**, 78
- Close Project**, 66
- Close Project Manager**, 67
- Compile *selected file***, 73
- Configuration**, 78
- Configure**, 67
- Connect**, 76
- Connection Organizer**, 76
- Contract Project**, 72
- Convert Legacy project**, 79
- Copy *selected file* as Link**, 69
- Copy *selected file* Local**, 69
- Customize Menus**, 79
- Debug Other Executable**, 77
- Debug *selected program***, 77
- Delete *selected file***, 69
- Edit**, 68
- Editor**, 78
- Exit All**, 67
- Expand Project**, 72
- File Names Only**, 72
- Filter Project View**, 73
- Flash *selected program***, 78
- Force link**, 75
- Full Paths**, 72
- Graphically Edit**, 68
- Highlight Selections**, 72
- Ignore build errors**, 75
- Ignore dependencies**, 75
- License Info**, 81
- Load Configuration**, 78
- Load Module**, 76
- Manuals**, 80
- Modify Project**, 68
- New Top Project**, 66
- New Window**, 67
- Only remove intermediate output**, 75
- Open Project**, 66
- Open Reference BSP Project**, 66
- Options**, 79
- Paste *selected file* as Link**, 69
- Paste *selected file* Local**, 69
- Preprocess *selected file***, 73
- Print Current View**, 66
- Print Expanded Project**, 66
- Project Manager Help**, 80
- Rebuild *selected file***, 73
- Recent Files**, 67
- Recent Projects**, 67
- Redo**, 69
- Relative Paths**, 72
- Reload Saved Project**, 66
- Remove *selected file***, 69
- Save Configuration As**, 78
- Save Configuration as User Default**, 78
- Save Project**, 66
- Search in All Source Files**, 72
- Search in *selected file***, 72
- Search in source file *inselected file***, 72
- Set Build Macros**, 70
- Set Build Options**, 67
- Set Build Target**, 69
- Set File Type**, 70
- Set Imported Environment Variables**, 70
- Set Options in Parent**, 70
- Show #includes Listing**, 75
- Show All Views**, 72
- Show build without execution**, 75
- Show internal commands**, 75
- Show Options Column**, 72
- Show Paths**, 72
- Show Size Column**, 73
- Show tool commands**, 75
- Show Type Column**, 72
- Simplify All Filenames**, 70
- Start Launcher**, 78
- Stop after cleaning output**, 75
- Stop Build**, 73

Index

- Task Window**, 76
- Troubleshooting Info**, 81
- Undo**, 69
- Use Utilities**, 79
- View Build Details**, 74
- View DoubleCheck Report**, 78
- Write Expanded Project to File**, 67
- Project Wizard**
 - using, 23
- projects
 - adding files to, 33
 - adding headers to, 36
 - adding libraries to, 36
 - building, 42
 - introduction to, 20
 - linking libraries to, 37
 - moving items in project hierarchy, 35
 - searching for files in, 46
 - setting options for, 62 to 63
- prototype_errors** driver option, 261
- prototype_silent** driver option, 261
- prototype_warnings** driver option, 261
- prototypes, 107
 - __PROTOTYPES__, 666
- provide linker expression function, 450
 - __PS_ABS() intrinsic function, 698
 - __PS_ADD() intrinsic function, 698
 - __PS_ADDS() intrinsic function, 699
 - __PS_DIV() intrinsic function, 698
 - __PS_DIVS() intrinsic function, 699
 - __PS_FDUP() intrinsic function, 699
 - __PS_MADD() intrinsic function, 698
 - __PS_MADDS() intrinsic function, 699
 - __PS_MADDS0() intrinsic function, 698
 - __PS_MADDS0F() intrinsic function, 699
 - __PS_MADDS1() intrinsic function, 698
 - __PS_MERGE00() intrinsic function, 698
 - __PS_MERGE01() intrinsic function, 698
 - __PS_MERGE10() intrinsic function, 698
 - __PS_MERGE11() intrinsic function, 698
 - __PS_MSUB() intrinsic function, 698
 - __PS_MSUBS() intrinsic function, 699
 - __PS_MUL() intrinsic function, 698
 - __PS_MULS() intrinsic function, 699
 - __PS_MULS0() intrinsic function, 698
 - __PS_MULS0F() intrinsic function, 699
 - __PS_MULS1() intrinsic function, 698
 - __PS_NABS() intrinsic function, 698
 - __PS_NEG() intrinsic function, 698
 - __PS_NEGS() intrinsic function, 699
 - __PS_NMADD() intrinsic function, 698
 - __PS_NMADDS() intrinsic function, 699
 - __PS_NMSUB() intrinsic function, 698
 - __PS_NMSUBS() intrinsic function, 699
 - __PS_RES() intrinsic function, 698
 - __PS_RESS() intrinsic function, 699
 - __PS_RSQRTE() intrinsic function, 698
 - __PS_SEL() intrinsic function, 698
 - __PS_SUB() intrinsic function, 698
 - __PS_SUBS() intrinsic function, 699
 - __PS_SUM0() intrinsic function, 698
 - __PS_SUM0F() intrinsic function, 699
 - __PS_SUM1() intrinsic function, 698
 - __PS_SUM1_F() intrinsic function, 699
 - __PS_SUM1F() intrinsic function, 699
 - __PS_SUM1FF() intrinsic function, 699
 - __PSQ_L() intrinsic function, 699
 - __PSQ_LX() intrinsic function, 699
 - __PSQ_ST() intrinsic function, 699
 - __PSQ_STX() intrinsic function, 699
- PT_DYNAMIC ELF program type, 823
- PT_HIPROC ELF program type, 824
- PT_INTERP ELF program type, 823
- PT_LOAD ELF program type, 823

Index

PT_LOPROC ELF program type, 824
PT_NOTE ELF program type, 823
PT_NULL ELF program type, 823
PT_PHDR ELF program type, 824
PT_SHLIB ELF program type, 824
__PTESYNC() intrinsic function, 693
__PTR_BIT, 670
__Ptr_Is_32, 677
__Ptr_Is_64, 677
__Ptr_Is_Signed__, 670
__Ptr_Is_Unsigned__, 670
putenv(), 736

Q

-Q driver option, 92
-Qn driver option, 246
Quit Building if Intex Warnings are Generated Builder option, 171
Quit Building if Warnings are Generated Builder option, 260
--quit_after_intex_warnings intex option, 171
--quit_after_warnings driver option, 260
quoted string assembler expression, 406
-Qy driver option, 246

R

-r assembler option, 395
-r librarian command, 476
r_addend ELF relocation field, 816
r_info ELF relocation field, 816
r_offset ELF relocation field, 816
r0 register
 using for ZDA, 138
RAM, placing variables in, 128
Raw Import Files Builder option, 246
-rawimport driver option, 246
.rc file extension, 41
--readonly_typeinfo driver option, 233

--readonly_virtual_tables driver option, 234
rebuild .gpj conditional control statement, 840
Rebuild *selected file* Project Manager menu item, 73
rebuilding
 Green Hills libraries, 751
Recent Files Project Manager menu item, 67
Recent Projects Project Manager menu item, 67
Recognition of Exported Templates Builder option, 238, 624, 641
Record Pathname (Not linked) Builder option, 255
Redirect Error Output to File Builder option, 260
Redirect Error Output to Overwriting File Builder option, 312
Redo Project Manager menu item, 69
reentrant functions, in C run-time libraries, 714
-ref assembler option, 395, 409
Reference Builder option, 285
:reference driver option, 285
__REG_BIT, 670
__Reg_Is_32, 677
__Reg_Is_64, 677
Register Allocation by Coloring Builder option, 289, 362
register allocation by coloring optimization, 362
register coalescing optimization, 363
Register Description File Builder option, 154
--register_definition_file driver option, 154
registers

Index

- allocator, 342
- I/O, 149
- implied usage of, 783
- size of, 677
- storing global variables, 134
- usage of, 105
- regs** assembler option, 394
- .rel** program section, 813
- .rela** program section, 813
- Relative Paths** Project Manager menu item, 72
- relative position of data objects, 777
- Reload Saved Project** Project Manager menu item, 66
- relobj** driver option, 244
- relocatable assembler expression, 406
- Relocatable Module Base Image** Builder option, 168
- Relocatable Module** Builder option, 167
- relocatable object files
 - creating, 244
 - debug information for, 244, 295
 - format, 802
 - relocation types, 815
- relocation directories
 - for ELF object files, 815
- relocation types, 815
- relprog** driver option, 244
- Remarks** Builder option, 259
- remarks** driver option, 259
- Remove selected file** Project Manager menu item, 69
- Rename Section** Builder option, 160
- repeat block directives, 420
- report viewer
 - gsar** Options, 374
 - launching from MULTI Project Manager, 373
 - launching from the command prompt, 373
- .rept** assembler directive, 421
- *_reserved** memory area, 441
- reserved symbols, 396
- restrict** driver option, 231
- restrict** keyword, 625
- restrict Keyword Support** Builder option, 231
- restrictions
 - memory optimization, 784
- Retain Comments During Preprocessing** Builder option, 301
- Retain Symbols for Unused Statics** Builder option, 228
- revision tracking, 427
- __RIR()** intrinsic function, 694
- __RLWIMI()** intrinsic function, 695
- __RLWINM()** intrinsic function, 695
- __RLWNM()** intrinsic function, 695
- .rodata** assembler directive, 422
- .rodata** data section, 458
- .rodata** directive, 460
- .rodata** program section, 127, 458, 461, 813
- ROM
 - putting data in, 458
- ROM linker section attribute, 453, 456
- .ROM** program section, 461
- ROM_NOCOPY linker section attribute, 453
- rpl_use_dbg_source_root** driver option, 163
- RTTI, 233
- __RTTI**, 676
- rtti** driver option, 233
- .run** output file type, 94
- run-time
 - copy and clear sections, 472
 - environment, 146
 - error checking, 190, 684

Index

- libraries, 704
- memory checking, 191
- type information
 - predefined macro, 676
- run-time code
 - customizing, 459
 - default, features that depend on, 761
 - ind_bcnt.c**, 758
 - ind_call.ppc**, 756
 - ind_crt1.c**, 757
 - ind_dots.ppc**, 756
 - ind_errn.c**, 757
 - ind_exit.c**, 759
 - ind_gcnt.ppc**, 758
 - ind_gpid.c**, 757
 - ind_gprf.c**, 758
 - ind_heap.c**, 758
 - ind_io.c**, 759
 - ind_iob.c**, 759
 - ind_lock.c**, 757
 - ind_manprf.c**, 757
 - ind_mcnt.ppc**, 758
 - ind_mprf.c**, 758
 - ind_reset.ppc**, 756
 - ind_stackcheck.c**, 757
 - ind_targ1.c**, 757
 - ind_thrd.c**, 758
 - libsys.a**, 459, 756
 - special program sections, 459
- run-time error checking
 - Builder and driver options for, 190
 - #pragma**, 684
- Run-Time Error Checks** Builder
 - option, 188, 684
- run-time memory checking
 - Builder and driver options for, 191
 - #pragma**, 684
- Run-Time Memory Checks** Builder
 - option, 191
- run-time type information, *see* RTTI

- Run-Time Type Information Support**
 - Builder option, 233
- RUNDIR context sensitive variable,
 - customization file, 860

S

- S** driver option, 90, 92, 238, 393
- .s** file extension, 40, 88
- S-Record format
 - generating, 244, 544
- Safe Commands to Execute After Associated Command (via Shell)**
 - Builder option, 283
- Safe Commands to Execute After Associated Command** Builder
 - option, 282
- Safe Commands to Execute Before Associated Command (via Shell)**
 - Builder option, 280
- Safe Commands to Execute Before Associated Command** Builder
 - option, 280
- saferc** driver option, 217 to 225
- Save Configuration As Project Manager**
 - menu item, 78
- Save Configuration as User Default**
 - Project Manager menu item, 78
- Save Project** Project Manager menu
 - item, 66
- .sbss** assembler directive, 427, 774
- .sbss** program section, 128, 138, 815
- .sbttl** assembler directive, 425
- scalar variables, 362
- .scall** assembler directive, 428
- Scan source files to augment native debug info** Builder option, 296
- scan_source** driver option, 296
- scxx** header file directory, 705
- SDA, *see* small data area optimization

Index

- sda driver option, 138, 158
- sda=0 driver option, 830
- .sdatabase program section, 461, 815
- %sdaoff() operator, 773
- .sdata assembler directive, 422
- .sdata program section, 128, 138, 815
- .sdata2 assembler directive, 422
- .sdata2 program section, 128, 138, 459
- Search for DBA** Builder option, 295
- Search in All Source Files** Project Manager menu item, 72
- Search in Files** dialog box, 46
- Search in Files Results** window, 47
- Search in selected file** Project Manager menu item, 72
- Search in source files in selected file** Project Manager menu item, 72
- Search Path for .dbo Files** Builder option, 295
- search_for_dba driver option, 295
- search_source_directory driver option, 302
- searching
 - for Builder options, 59
 - for files in your project, 46
 - in files, 46
- searching libraries automatically, 708
- .secinfo program section, 462, 814
- SECOND_PASS_OPTION context sensitive variable, customization file, 860
- .section assembler directive, 423
- section driver option, 161
- section maps, 442
 - See also* linker directives files (.ld)
- Section Overlap Checking** Builder option, 257
- section_prefix driver option, 304
- section_suffix driver option, 304
- sections
 - control directives, 421
 - defining, 442
 - #pragma directive, 129
 - SECTIONS linker directive, 442
 - :select Builder option, 48, 284
 - 'Select One' Project Extension List** Builder option, 284
 - selectone_optional .gpj conditional control statement, 840
 - selectone_required .gpj conditional control statement, 840
 - Self-Dependency** Builder option, 277
 - .set assembler directive, 401, 427
 - Set Build Macros** Project Manager menu item, 70
 - Set Build Options** Project Manager menu item, 67
 - Set Build Target** Project Manager menu item, 69
 - Set File Type** Project Manager menu item, 70
 - Set Imported Environment Variables** Project Manager menu item, 70
 - Set INTEGRITY Distribution** menu item, 78
 - Set Maximum DBO Not Found Warnings** Builder option, 296
 - Set Message to Error** Builder option, 264
 - Set Message to Remark** Builder option, 265
 - Set Message to Silent** Builder option, 265
 - Set Message to Warning** Builder option, 264
 - Set Options in Parent** Project Manager menu item, 70
 - setenv(), 736
 - __setflm() intrinsic function, 694
 - __SETSR() intrinsic function, 695
 - settings for stand-alone programs

Index

- Project Manager, 31
- settings for your project
 - Project Manager, 31
- sh_addr ELF section header field, 809
- sh_addralign ELF section header field, 810
- sh_entsize ELF section header field, 810
- sh_flags ELF section header field, 809, 811
- sh_info ELF section header field, 810, 812, 817
- sh_link ELF section header field, 810, 812, 817
- sh_name ELF section header field, 809, 821
- sh_offset ELF section header field, 809
- sh_size ELF section header field, 809, 821
- sh_type ELF section header field, 809, 812
- SHA-1** Builder option, 171
- sha1** intex option, 171
- Shadow Declarations** Builder option, 263
- Shared Libraries** Builder option, 162
- SHF_ALLOC ELF section attribute flag, 812
- SHF_EXECINSTR ELF section attribute flag, 812
- SHF_GHS_ABS ELF section attribute flag, 812
- SHF_WRITE ELF section attribute flag, 812
- SHFLAGS linker section attribute, 453
- shift operators, 782
- SHN_ABS ELF section index, 810
- SHN_COMMON ELF section index, 810
- SHN_GHS_SMALLCOMMON ELF section index, 810
- SHN_GHS_TINYCOMMON ELF section index, 810
- SHN_GHS_ZERO_COMMON ELF section index, 810
- SHN_UNDEF ELF section index, 810, 812
- .short assembler directive, 418
- short data type, 106
 - size of, 670
 - variable arguments, 596
- short_enum** driver option, 107, 226, 595
- shortcut commands
 - backward, 46
 - forward, 46
- Show #includes Listing** Project Manager menu item, 75
- Show All Views** Project Manager menu item, 72
- Show build without execution** Project Manager menu item, 75
- Show internal commands** Project Manager menu item, 75
- Show Options Column** Project Manager menu item, 72
- Show Paths** Project Manager menu item, 72
- Show Size Column** Project Manager menu item, 73
- Show tool commands** Project Manager menu item, 75
- Show Type Column** Project Manager menu item, 72
- __SHRT_BIT, 670
- .shstrtab program section, 813
- SHT_NOTE ELF section type, 811
- SHT_NULL ELF section type, 811
- SHT_PROGBITS ELF section type, 811
- SHT_REL ELF section type, 811 to 812
- SHT_RELA ELF section type, 811 to 812
- SHT_STRTAB ELF section type, 811

Index

- SHT_SYMTAB ELF section type, 811 to 812
- .si** files, 600
- SIBLINGS context sensitive variable, customization file, 860
- SIBLINGS[*type*] context sensitive variable, customization file, 860
- signed keyword, 583
- signed_chars** driver option, 225, 609
- `__SIGNED_CHARS__`, 670
- signed_fields** driver option, 225, 592
- signed_pointer** driver option, 226
- Signedness of Bitfields** Builder option, 225, 587, 592, 659
- Signedness of Char Type** Builder option, 225, 587, 609, 658
- Signedness of Pointers** Builder option, 225
- SIMD vectorization, 343
- Simplify All Filenames** Project Manager menu item, 70
- Simplify C Print Functions** Builder option, 288
- single-pass inlining, 320
- single-thread build mode, setting, 47
- `.size` assembler directive, 428
- `sizeof` linker expression function, 450
- `.skip` assembler directive, 418
- slash_comment** driver option, 196
- small data area optimization
 - assembler directives, 140, 422
 - automatic link-time allocation, 141, 158
 - base register, 137
 - builder and driver options for, 138, 141, 145, 158, 269
 - excluding data from, 140
 - introduction to, 137
 - linker directives file for, 142
 - linker errors, 145
 - linking modules, 144
 - `#pragma` directives, 139, 688
 - predefined macros, 673
 - program sections for, 128, 138, 815
 - RAM SDA, 137
 - register pointers, 105
 - ROM SDA, 137
 - setting a threshold value for, 138, 158, 673
 - specifying additional registers for, 145, 269
- small** driver option, 185
- `__SoftwareDouble__`, 672
- `__SoftwareFloat__`, 672
- source code
 - interleaving with assembly code, 242
- Source Directories Relative to This File** Builder option, 174
- Source Directories Relative to Top-Level Project** Builder option, 276
- Source Directory Is Include Directory** Builder option, 302
- source files
 - setting options for, 63
- source files for customization, 146
- source level debugging
 - problems, 785
- source listing, 395
- Source Listing Generation** Builder option, 125, 241 to 242
- Source Listing Generation Output Directory** Builder option, 241
- Source Root** Builder option, 163, 172
- :sourceDir** Builder option, 174
- :sourceDirNonRelative** Builder option, 276
- `.space` assembler directive, 418
- SPE** assembler option, 394

Index

- SPE driver option, 271
- __SPE__, 668
- Special Data Area** Builder option, 138, 140, 158, 688
- specifying individual functions to
 - inline, 322
- spl instruction, 789
- %SPOFF, 795
- sqrt(), ANSI C implementation, 615
- sqrtf(), ANSI C implementation, 615
- srec driver option, 245
- st_info ELF symbol table entry, 818
- st_name ELF symbol table entry, 818
- st_other ELF symbol table entry, 819
- st_shndx ELF symbol table entry, 819 to 820
- st_size ELF symbol table entry, 818
- st_value ELF symbol table entry, 818, 820
- stabs2dbo driver option, 299
- stack
 - generating frame to support traces, 293
 - pointer, 105
 - usage, 107
- Stack Limit Checking** Builder option, 160
- .stack program section, 463, 815
- stack_check driver option, 160
- Standard C++, 624
- standard I/O package, 709
- Standard Include Directories** Builder
 - option, 301
- __STDC__, 667
- __STDC_CXX_HEADERS, 667
- standard_vtbl driver option, 234
- Start Address Symbol** Builder
 - option, 147, 245, 437
- Start File Directory** Builder option, 252
- Start Files** Builder option, 252
- Start Launcher** Project Manager menu
 - item, 78
- start-up module
 - rebuilding, 751
- startfile_dir driver option, 253
- startfiles driver option, 252
- startup code
 - crt0.o, 459, 753
 - default, features that depend on, 761
 - ind crt0.c, 755
 - ind_mcpy.ppc, 755
 - ind_mset.ppc, 755
 - ind_reset.ppc, 755
 - libstartup.a, 459, 755
- static
 - constructors and destructors, 655, 765, 768
- static address elimination
 - optimization, 361
- static analysis, *see* DoubleCheck
- STB_GLOBAL ELF symbol binding, 819
- STB_HIPROC ELF symbol binding, 819
- STB_LOCAL ELF symbol binding, 819
- STB_LOPROC ELF symbol binding, 819
- STB_WEAK ELF symbol binding, 819
- std driver option, 195, 624
- STD driver option, 195, 624
- std namespace, 675
- std_cxx_include_directory driver
 - option, 302
- <stdarg.h>, 595, 597, 779
- __STDC__, 665
- __STDC_HOSTED__, 665
- __STDC_VERSION__, 665
- stderr, 766
- stderr driver option, 260
- stderr=errfile librarian option, 478
- stderr_overwrite driver option, 312
- stdin, 766
- stdinc driver option, 302

Index

- stdio**, 709
- stdl** driver option, 196, 667
- stdle** driver option, 196, 667, 706
- stdlib** driver option, 310
- stdlib.h**
 - functions, 749
- stdout, 766
- sterror(), ANSI C
 - implementation, 619
- __sthbrx() intrinsic function, 695
- Stop after cleaning output** Project Manager menu item, 75
- Stop Build** Project Manager menu item, 73
- strength reduction optimization, 340
- strftime(), ANSI C
 - implementation, 615
- __STRICT_ANSI__, 666
- strict_overlap_check** driver option, 257
- string constants, 397
- string literals, 626
- string tables, 821
- strings not referenced, 821
- strip** driver option, 245
- strip_application intex** option, 171
- .strtab program section, 814
- struct data type, 583, 591, 625
 - packing, 674
 - portability, 778
- structure packing, 108
 - packing all structures, 111
 - packing one structure type, 110
 - pointing to packed structures, 111
 - portability, 111
 - warnings, 111
- .strz assembler directive, 418
- STT_FILE ELF symbol type, 820
- STT_FUNC ELF symbol type, 819
- STT_HIPROC ELF symbol type, 820
- STT_LOPROC ELF symbol type, 820
- STT_NOTYPE ELF symbol type, 819
- STT_OBJECT ELF symbol type, 819
- STT_SECTION ELF symbol type, 820
- __stwbrx() intrinsic function, 695
- __STWCX() intrinsic, 692
- subprojects
 - setting options for, 62
- subroutines, calling, 770
- .subtitle assembler directive, 425
- Support __noinline Keyword Builder** option, 231, 329
- Support floating point types in scanf** Builder option, 310
- Support for C Type Information in Assembly** Builder option, 243
- Support for Constructors/Destructors** Builder option, 232
- Support for Implicit Extern C Type Conversion** Builder option, 308
- Support for Implicit Typenames** Builder option, 236
- Support for Old-Style Specializations** Builder option, 235 to 236
- Support for Very Large Switch Statements** Builder option, 273
- Suppress Dependencies** Builder option, 278
- suppress_vtbl** driver option, 234
- switch statements
 - MISRA C 2004, 199
- symbol tables, 818
- symbolic
 - debugging directives, 427
 - memory-mapped I/O, 148
- symbols
 - bindings, 819
 - definition directives, 426
 - definition of, 463
 - types, 819

Index

- values, 820
- Symbols Referenced Externally** Builder
 - option, 181
 - .symtab program section, 814
 - __SYNC() intrinsic function, 693
- syntax** driver option, 92
- sys_include_directory** driver option, 302
 - .syscall program section, 463, 756, 814
- syslib** driver option, 162
- System Include Directories** Builder
 - option, 302
- T**
- T** driver option, 247
- t** librarian command, 476
- T** linker option, 435
- tail call optimization, 356
- Tail Calls** Builder option, 290, 356
- target
 - specifying, 71
- Target Processor** Builder option, 269
- target resources project, 28
- target!=target_name .gpj** conditional control statement, 840
- target-based timing profiling
 - debugging information, 120
- target=target_name .gpj** conditional control statement, 840
- Task Window** Project Manager menu
 - item, 76
- Tektronix hexadecimal format
 - generating, 244, 544
- Template Instantiation Output Directory** Builder option, 237
- templates
 - builder and driver options for, 234 to 238
 - class, 634
 - dependent name processing, 238
 - entity, 634
 - exported templates, 238, 641
 - function, 634
 - function inlining, 329
 - guiding declarations, 235
 - implicit inclusion of, 236, 640
 - instantiation and namespaces, 645
 - instantiation directory, 237
 - instantiation of, 634
 - link-once instantiation of, 637
 - #pragma directives, 638
 - prelinker, 238, 635
 - prelinker instantiation, 635
 - specialization, 634, 636
- temporary labels, 407
- Temporary Output** Builder option, 275
- Temporary Output Directory** Builder option, 274
 - .text assembler directive, 423
 - .text program section, 127, 458, 814
 - __thread keyword, 590
- thread-local storage, 590
- thread-safety
 - customizing libraries, 720
- thread-safety library functions
 - library functions, 714
- __THREADX, 676
- .ti** file, 635
- .ti** output file type, 93
- .time** output file type, 94
- __TIME__, 583, 665
 - precompiled header files and, 651
- timer_profile** driver option, 121, 188
- .title assembler directive, 425
- __TLBSYNC() intrinsic function, 693
- tmp** driver option, 274
- tmp=dir** librarian option, 478

Index

- `tmpfile()`, ANSI C
 - implementation, 617
- Top Project, 28
 - creating, 22
- Top Project directives, for .gpj files, 835
- TOPPROJECT context sensitive variable,
 - customization file, 860
- Toyota Motor Corporation, 587
- Transition C
 - variable arguments, 595
- Treat Doubles as Singles** Builder
 - option, 155
- Treatment of RTTI as const** Builder
 - option, 233
- Treatment of Virtual Tables as 'const'**
 - Builder option, 234
- Trigraphs** Builder option, 263
- Troubleshooting Info** Project Manager
 - menu item, 81
- two-pass inlining, 321
- TX_ENABLE_EVENT_LOGGING, 676
- .txt file extension, 41
- .type assembler directive, 428
- type combinations, 406
- type qualifiers, 588
 - treatment in GNU C, 576
- typographical conventions, xxii

- U**
- u driver option, 254
- U driver option, 192
- UA driver option, 255
- unalign_debug_sections linker
 - option, 435
- unaligned loads and stores, handling, 111
- unary, 401
- Undefine Preprocessor Symbol** Builder
 - option, 192
- undefined assembler expression, 406
- undefined behavior, 606
- undefined driver option, 253
- Undefined Preprocessor Symbols** Builder
 - option, 264
- Undefined Symbols** Builder option, 253
- underscore linker option, 435
- Undo** Project Manager menu item, 69
- union data type, 583, 591, 625
 - portability, 778
- unique_strings driver option, 228
- Uniquely Allocate All Strings** Builder
 - option, 228, 657
- __unix_asm, 668
- Unknown Pragma Directives** Builder
 - option, 262, 679
- unknown_pragma_errors driver
 - option, 262, 679
- unknown_pragma_silent driver
 - option, 262, 679
- unknown_pragma_warnings driver
 - option, 262
- Unroll Larger Loops** Builder option, 51, 180, 342
- unsafe_predefines driver option, 300, 664
- unsaved DoubleCheck property
 - type, 372
- unsigned int __MFTB(void);
 - intrinsic, 692
- unsigned_chars driver option, 225, 587, 609
- unsigned_fields driver option, 225, 587, 592
- unsigned_pointer driver option, 226
- unspecified behavior, 606
- Use Floating-Point in stdio Routines**
 - Builder option, 157

Index

Use Smallest Type Possible for Enum

Builder option, 107, 226, 588, 595, 658

Use Source Root for RPX/RPL Builder

option, 163

Use Utilities Project Manager menu

item, 79

-use_fsel driver option, 270

--use_pch driver option, 648

user-defined variable, 458

--using_std driver option, 230, 675

utility programs, 482

gaddr2line, 484

gasmlist, 485

gbin2c, 487

gbincmp, 496

gbldconvert, 498

gbuild, 499

gcolor, 506

gcompare, 511

gdump, 514

gfile, 521

gfunsize, 522

ghexfile, *see* utility programs, **gsrec**

ghide, 523

gmemfile, 524

gnm, 531

grun, 540

gsize, 542

gsrec, 544

gstack, 551

gstrip, 554

gversion, 555

gwhat, 557

-uvfd driver option, 251

V

-V assembler option, 395

-v driver option, 95

-V driver option, 260

-v linker option, 435

-V linker option, 436

values

representable range, 776

<varargs.h>, 595 to 596

variable

alignment, 680

variable arguments, 595

variables

accessing, 770

allocation, 785

local, automatic allocation, 363

placement of for embedded

development, 128

__VEC__, 668

vector, 669

vector data type, 106

Vector Peeling Builder option, 292, 350

vectorization, 343

version number

generating from assembler, 395

generating from compiler driver, 260

View Build Details Project Manager menu

item, 74

View DoubleCheck Report Project

Manager menu item, 78

viewing reports

DoubleCheck, 373

Virtual Function Definition Builder

option, 234

.vle assembler directive, 429

VLE Code Generation and Libraries

Builder option, 162

-vle driver option, 162

__vle__, 669

void data type, 583

volatile keyword, 360, 583, 590, 785

volatile registers, 105

Index

W

-w assembler option, 395

-w driver option, 259, 607

-w linker option, 436

Wait for Debug Translation Before

Exiting Builder option, 297

 __WAIT__() intrinsic function, 693

-wait_for_dblink driver option, 297

-warn_dbo_not_found_max driver option, 296

.warning assembler directive, 429

%warning preprocessor directive, customization file, 863

Warnings Builder option, 259, 607

--warnings driver option, 259

__WCHAR_BIT__, 670

wchar.h

 functions, 749

 __WChar_Is_Int__, 677

 __WChar_Is_Long__, 678

 __WChar_Is_Short__, 678

 __WChar_Is_Signed__, 670

 __WChar_Is_Unsigned__, 670

 _WCHAR_T, 676

wchar_t type

 predefined macro, 676

 predefined macros, 670

 size, 670

.weak assembler directive, 427

weak symbols

 defining, 427, 682

 ELF symbol binding, 819

 resolving from libraries, 434, 466

 testing for in linker directives, 450

-Wformat driver option, 263

whitespace, 400

whole program optimizations

 constant-value global variables,

 deletion of, 336

 constant-value global variables, propagation of, 335

wholeprogram optimizations

 constant function returns, deletion of, 335

 constant parameters, deletion of, 335

 function parameters, constant

 propagation of, 335

 one-site inlining, 334

 unreachable functions, deletion of, 336

-Wimplicit-int driver option, 263

-Wl, driver option, 249

-Wno-format driver option, 263

-Wno-implicit-int driver option, 263

-Wno-shadow driver option, 263

-Wno-trigraphs driver option, 264

-Wno-undef driver option, 264

word size, 775

workspaces

 overview, 5

-wrap sym linker option, 436

--wrap_diagnostics driver option, 261

Write Expanded Project to File Project

 Manager menu item, 67

-Wshadow driver option, 263

-Wtrigraphs driver option, 264

-Wundef driver option, 264

X

-X driver option, 313

-x librarian command, 476

X-Switches Builder option, 313

--xref=declare driver option, 297

--xref=full driver option, 297

--xref=global driver option, 297

--xref=none driver option, 297

Index

Y

- Y0, driver option, 286
- Ya, driver option, 287
- Yl, driver option, 287
- YL, driver option, 286

Z

- ZDA, *see* zero data area optimization
- zda driver option, 138, 158
 - .zdata program section, 815
- zero data area optimization, 138

- builder and driver options for, 138, 158
- linker directives file for, 143
- linker errors, 145
- linking modules, 144
- #pragma directives, 688
- predefined macros, 673
- program sections for, 128, 138, 815
- setting a threshold value for, 158, 673
- size of, 138
- zero-initialized variables, 774